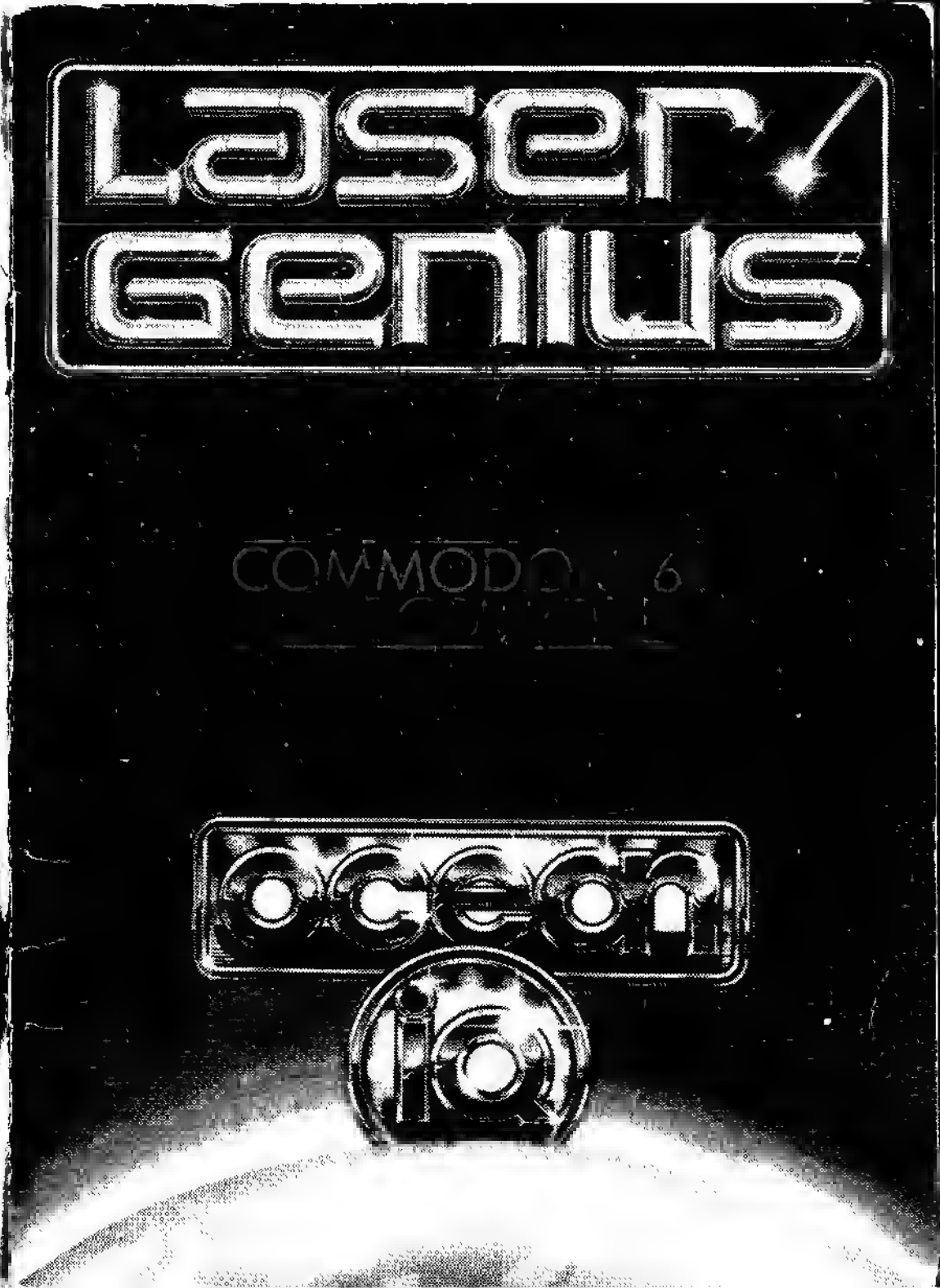
Laser Genius (Ocean)
====================

This is an assembler/monitor package. The label on the original disk says:

COMMODORE 64
   LASER/
   GENIUS
   ocean
    iq

```
LASER GENIUS.d64
----------------
0 "LASER GENIUS     " ID 2A
65    "ASML"              PRG
2     "ASM"               PRG
65    "ASME"              PRG
3     "DLOAD"             PRG
2     "GENASM"            PRG
2     "GENMON"            PRG
2     "RELMON"            PRG
2     "LOADER"            PRG
53    "MON"               PRG
70    "RMON"              PRG
398 BLOCKS FREE.
```



GENIUS
by OASIS SOFTWARE

THIS MANUAL

Piracy has reached epidemic proportions and it is with regret that we are forced to reproduce this manual in a form which cannot be photocopied. Our apologies for the inconvenience this may cause to our genuine customers. A reward will be paid for information leading to the successful prosecution of parties infringing this Copyright Notice.

NOTE

This manual is essential for the use of Genius. For this reason we would warn customers to look after it very carefully, as separate manuals will not be issued under any circumstances whatsoever.

ENQUIRIES

If you have any queries on the use of the Genius package, please send them to us in a letter, ensuring you enclose the Technical Enquiry Card printed at the back of this manual. A new card will be returned to you with your reply. Please note that telephone enquiries, and enquiries not accompanied by the card, cannot be answered.

CONTENTS

64 - MAC

GENIUS FOR THE COMMODORE 64

by David Hunter, Martin Lewis and Andrew Foord

Genius is a complete machine code development system for the Commodore 64 and comprises three compatible parts. The first two parts, 64-MAC and 64-MON, load as one file and the third, the monitor/analyser, loads separately. Many of the 64-MON features are replicated by the monitor-analyser but the latter also has a number of very powerful extensions, in particular the analyser.

64-MAC/MON

64-MAC/MON provides a comprehensive set of over 70 commands for writing and debugging assembly language programs on the COMMODORE 64. It includes a line editor for the creation of source text, a full two-pass macro assembler, a symbolic disassembler, a machine code monitor and a tracer.

The editor automatically checks the syntax of lines as they are typed in, and formats the source text when it is listed. It includes block delete, move and copy as well as search and replace commands and automatic line numbering.
The assembler can be operated in either 'resident' or 'disk' mode. Resident mode is ideal for learning about assembly language or writing small programs - assembly is extremely fast, at over 20,000 lines per minute. Because text is tokenised when in memory, programs of over 2,500 lines can be written without having to use disk mode. In disk mode, linked files on floppy disk may be assembled. The size of program that can be written in this manner is only limited by the amount of mass storage available; about 8,500 lines of code in the case of the 1541 single floppy disk. The assembler also includes conditional assembly, cross-referencing and a printer pagination facility.

The machine-code monitor commands allow direct inspection and modification of memory - commands to list, move, relocate, compare, modify, search and disassemble blocks of memory are included. Up to 16 blocks which are printed as .BYTE, .WORD or .DBYTE directives when disassembling may be defined.
The tracer can single step through a machine-code program, displaying the register contents and the contents of up to 16 memory locations after executing each instruction. Options exist to suppress single stepping and register printing or to print only the program counter. Up to 16 locations can be defined at which the registers are always printed, even if register printing is disabled.

Two copies of 64 MAC/MON are supplied with the disk version - one is located in low memory and one in high memory. Only the low memory version is supplied on tape.

The low memory version occupies memory from $0800 to $47FF. The BASIC ROM from $A000 to $BFFF is switched out of memory after 64 MAC/MON has loaded, or after using RUN/STOP-RESTORE. However, 64 MAC/MON will still operate correctly if you re-enable the ROM by setting bit 0 of location 1.

The high memory version resides from $9000 to $CFFF. The BASIC ROM is enabled whenever memory is accessed by one of the monitor commands.

Note that in both cases, full use of the zero page by the user's programs is allowed.

1. LOADING

1.1 LOADING FROM DISK

After switching on the computer system, insert the floppy disk into the drive and type the following:

```
LOAD "GENASM",8,1
```

After about ten seconds you are asked to press "L" or "H" to select between the low and high memory versions. Once you have done this it takes approximately one minute to load.

1.2 LOADING FROM TAPE

The programs are arranged on the tapes as follows:

Tape 1: **Side A:** The standard monitor/analyser program which resides at 3000 decimal

Side B: A relocatable version of the monitor/analyser program.

Tape 2: **Side A:** Assembler/Monitor (Turbo Loading) "GENASM"
Object Code Loader "LOADER"

Side B: Assembler/Monitor (Normal Loading) "GENASM"

To load in the assembler, switch the machine on and insert Tape 2 into the tape unit at either Side A or Side B. Press SHIFT RUN/STOP and press PLAY in the usual way.

1.3 MEMORY ALLOCATION

When 64-MAC/MON has loaded, the following message is printed:

```
64-MAC/MON V1.5L

COPYRIGHT 1986 DAVID HUNTER

NEW (Y/N)? Y
TEXT MEMORY?
```

Unless you wish to reserve memory for your own machine code routines, hit RETURN - this reserves memory from $4800 to $CFFF for use as text storage, giving 34816 bytes free. Otherwise, type in the lower and upper limits of memory to be used, separated by a comma.

A 'BYTES FREE' message is then printed, followed by the READY prompt.

Note that the computer's memory is completely cleared after loading, but it remains unaltered after subsequent NEW commands.

The following locations in pages 2 and 3 are altered by 64-MAC/MON:

$0200 to $0258 used for temporary storage.
$0314 to $0315 IRQ re-vectored for function keys.
$0316 to $0317 BRK
$0318 to $0319 NMI re-vectored for RUN/STOP-RESTORE.

2. 6502 ASSEMBLY LANGUAGE

This section is not intended to teach assembly language programming - if you are a novice to the subject, we suggest that you read '6502 Assembly Language Programming' by Lance A. Leventhal, which is published by McGraw-Hill. Another worthwhile text is '6502 Assembly Language Subroutines' by Leventhal and Saville, published by Osborne/McGraw-Hill. However, the information presented here should suffice if you have knowledge of another microprocessor.

2.1 NOTATION

2.1.1 A <label> consists of a letter followed by up to fourteen of the following characters:

'A'..'Z' , '0'..'9' , '_' , '.' , '$'

Examples: COMPARE$NAMES OUTPUT3 T9

2.1.2 A <numerical constant> consists of one of the following:

"%" followed by a binary number
"@" followed by an octal number
a decimal number
"$" followed by a hexadecimal number
"'" followed by an ascii character (followed by an optional second quote)

Examples: %00001101 $ACD9 '7 19 '&'

2.1.3 An <expression> consists of <numerical constant>s and/or <label>s separated by the following operators:

"+" add
"-" subtract
"*" multiply
"/" divide
"?" exclusive-or
"&" logical and

There is no operator precedence, and brackets may not be used (this only applies to <expression>s that are included as part of an assembly language program). If '<' is placed before an expression, it is converted to a '&255' at the end of the expression when printing; similarly, '>' is converted to '/256'.

Examples: NUMBER$BASE+3 <INTERPRETER-1 INPUT$BUFFER/256
'Z'+1

2.1.4 A <string constant> is a number of ASCII characters enclosed in single quotes. If one of the characters is to be a quote then two successive quotes must be used.

Examples: 'BPLBMIBCCBCSBNEBEQBVCBVS' '' '$@"%'

2.2 ASSEMBLY LANGUAGE STATEMENTS

There are three types of assembler statements: directives, instructions and comments.

2.2.1 Directive Statements

These may be considered as instructions which are obeyed at assembly time rather than run time. A directive statement consists of the following:

<label> <directive> <operand> <comment>

The label and comment fields are optional, and the operand field is not required in some cases. This assembler supports 27 directives, details of which are given below:

2.2.1.1 .BYTE directive

This is used to define single-byte constants. It should be followed by a number of <expression>s and/or <string constant>s separated by commas.

Examples:

```
POWERSOF2  .BYTE  1,2,4,8,16,32,64,128
HEXCHARS   .BYTE  '0123456789ABCDEF',0
```

2.2.1.2 .DBYTE directive

This has the same syntax as .BYTE but it generates two-byte constants in high-byte/low-byte order.

2.2.1.3 .WORD directive

This is the same as .DBYTE but the constants are in low-byte/high-byte order.

2.2.1.4 .PAD directive

The .PAD directive is used to pad out an area of program with NOP bytes.

Examples: .PAD *&$FF00+256-*
.PAD 6

2.2.1.5 .END directive

This is used to mark the end of an assembly language program. It is optional if the assembler is being used in resident mode.

2.2.1.6 .BLOCK directive

This directive is used to reserve space - it is followed by an expression which is added to the location counter.

Examples:
```
INPUT$BUFFER        .BLOCK 72
XPOS                .BLOCK 2
```

2.2.1.7 (equals) directive

The '=' directive is used to equate a label to an <expression>.

Examples:
```
INTERRUPTPERIOD=3906/SAMPLERATE
CR             =13
```

It is important to realise that these calculations are carried out at assembly time, not run-time.

2.2.1.8 '*'

'*' is a reserved symbol which refers to the location counter during assembly. The program location counter may be set like this: *=$9000, and blocks of memory may also be reserved:

```
INPUT$BUFFER  *=*+72
```

2.2.1.9 .ORG directive

This is used to set the program location origin.

Example:
```
.ORG $9000
```

Although this appears to be the same as *=$9000, there is a subtle difference between them which is explained in section 9.3.

2.2.1.10 .DEFMAC directive

This directive should be placed at the start of a macro definition. The label preceding the directive defines the macro name. It should be followed by a list of formal parameter labels separated by commas.

2.2.1.11 .ENDMAC directive

.ENDMAC is used at the end of a macro definition.

2.2.1.12 Macro Invocations

To call a macro in the program body, its name should be preceded by a colon and followed by a list of actual parameter expressions separated by commas.

Example...
```
1230 OUTPUT         .DEFMAC  START,MODE
1240                LDA  #START&255
1250                LDY  #START/256
1260                LDX  #MODE
1270                JSR  PRINT
1280                .ENDMAC
.
.
.
3450                BNE  LOOP
3460                :OUTPUT  ALPHA+6,3
.
.
.
```


In resident mode, macros can be defined anywhere in the text either before or after they are used, although it is best to keep them near the top of the program as this speeds up assembly. When assembling programs in disk mode, all macro definitions must be in the first file.

If a symbol is defined inside a macro and the macro is called more than once then a 'label defined twice' error message will be printed. To circumvent this problem, use the '*' symbol as in the following example:

```
.
.
.
100  DELAY          .DEFMAC  DEL
110                 LDX  #DEL
120                 DEX
130                 BNE  *-1
140                 .ENDMAC
.
.
.
.
960                 :DELAY  10
.
.
990                 :DELAY  20
.
.
.
```

2.2.1.13 .IFEQ directive

If the expression following this directive is zero, assembly continues as normal, otherwise code generation is suppressed until the next .ELSE or .IFEND.

2.2.1.14 .IFNEQ directive

If the expression following this directive is non-zero, assembly continues as normal, otherwise code generation is suppressed until the next .ELSE or .IFEND.

2.2.1.15 .IFPOS directive

If the expression following this directive is in the range 1 to 32767, assembly continues as normal, otherwise code generation is suppressed until the next .ELSE or .IFEND.

2.2.1.16 .IFNEG directive

If the expression following this directive is in the range 32768 to 65535, assembly continues as normal, otherwise code generation is suppressed until the next .ELSE or .IFEND.

2.2.1.17 .IFEND directive

This is used at the end of a conditional assembly .IF construct assembly after it, proceeds as normal.

2.2.1.18 .ELSE directive

This works like the ELSE statement in extended BASICs- if code generation is suppressed, it is enabled, and vice-versa.

2.2.1.19 Examples of conditional assembly:

```
17450  OUTPUT   .IFEQ  CBM64
17460           JSR  $FFD2    ;CBM64 OUTPUT ROUTINE
17470           BCS ERROR1
17480           .ELSE
17490           STX  TEMP
17500           TAX
17510           JSR  $0238    ;ORIC ATMOS OUTPUT ROUTINE
17520           LDX  TEMP
17530           PHA
17540           LDA  KEYCHAR
17550           CMP  #$83     ;CONTROL-C?
17560           BEQ  ERROR2
17570           PLA
17580           .IFEND
```

```
2750                LDA  #PRINTERON&255
2760                LDY  #PRINTERON/256
2770                BIT  PRINTERFLAG
2780                BMI  PRINTMESSAGE
2790  PRHI          =PRINTEROFF/256
2800                .IFNEQ  PRINTERON/256-PRHI
2810                LDY  #PRHI
2820                .IFEND
2830                LDA  #PRINTEROFF&255
2840  PRINTMESSAGE  JSR  OUTPUT$MESSAGE
```

2.2.1.20 .PRINT directive

This directive should be followed by a <string constant> which is simply printed when the directive is encountered during assembly.

Example:

```
19270  MESSAGES
19280  MSG1      .BYTE  '?SYNTAX  ERROR',0
19290  MSG2      .BYTE  'NUMBER TOO BIG',0
```

```
19440  MSG17     .BYTE  'FOUND ',0
19450  MSGEND
19460            .IFNEQ  MSGEND-MESSAGES/256
19470            .PRINT  'MESSAGE TABLE IS LONGER
                  THAN 256 BYTES'
19480            .END
19490            .IFEND
```


2.2.1.21 .LIST directive

This directive turns on the generation of an assembler listing, except if object code is being assembled to disk or tape.

2.2.1.22 .NOLIST directive

This turns off the generation of an assembler listing.

2.2.1.23 .PAGE directive

If an assembler listing is being output to the printer, this directive will start a new page.

2.2.1.24 .PAGEIF directive

This should be followed by an <expression> - if this is greater than the number of lines left on the page, a new page is taken, otherwise one line is skipped. This only takes place if an assembler listing is being output on the printer.

Example: `.PAGEIF 24`

2.2.1.25 .SKIP directive

This is used to print a certain number of blank lines when assembling a listing to the printer.

Example: `.SKIP 2`

If the number is left out, a default value of 1 is assumed.

2.2.1.26 .TITLE directive

This should be followed by a <string constant> which will be printed at the top of each new page on the printer.

Example: `.TITLE 'C64 MACRO ASSEMBLER'`

2.2.1.27 .WIDTH directive

This sets the number of characters printed per line on the printer.

Example: `.WIDTH 96`

2.2.1.28 .HEIGHT directive

This sets the number of lines printed per page on the printer.

Example: `.HEIGHT 66`

2.2.1.29 .INTNUM directive

This initialises the printer page number to zero.

2.2.1.30 .FILE directive

This directive is used to link files together in disk mode. At the end of each file, there should be a .FILE directive followed by a <string constant> consisting of the name of the next disk file.

Example: `.FILE 'ASM4'`

2.2.1.31 All directives, apart from .PAGE, .END and .ENDMAC may be abbreviated to their first three letters.

Example: `.BYT $C9,$A9,$89`

2.2.2 Instruction statements

An instruction statement consists of:

<label> <opcode mnemonic> <operand> <comment>

The <label> and <comment> fields are optional. Details of the allowable <opcode mnemonic>s and <operand>s are given in sections 16 and 17 of this manual respectively.

If during the first pass of assembly an instruction which has both zero page and absolute addressing modes has as its operand an undefined expression, as in this example:

```
10                *12345
20                LDA  FIVE
30  ABCDEF        JMP  ABCDEF
40  FIVE        =5
```

and the expression is evaluated during the second pass as being less than 256, the assembler will insert an extra NOP byte before the next label definition during the second pass.

2.2.3 Comment statements

A comment statement consists of the following:

; <comment>

The <comment> may be any commentary whatsoever.

3. ARITHMETIC EXPRESSIONS IN COMMAND MODE

3.1 In command mode, line numbers, memory locations and so on are expressed as <expression>s, as defined in 2.1.3, with the difference that brackets may be used and operator precedence exists. A '#' is used to represent the logical-or operator. A full stop may be used to represent the last result from the CALC command, and !<label> gives the line number in which a label is defined.

3.2 A <string> is defined as a series of characters bounded by one of the following delimeters:

! " # $ % & ' ( ) * + , – . /

If the <string> is to be followed by an end-of-line, the delimeters may be omitted.

4. USING THE EDITOR

4.1 The screen editor may be used as in BASIC. The RUN/STOP key terminates a listing at any time. CTRL slows down printing and the SPACE bar can be used to temporarily halt a listing- pressing it again restarts the listing.

4.2 FUNCTION KEYS

The function keys may be defined as follows, where n is the number of the function key:

```
Fn=<string>
```

The back-arrow key at the top left of the keyboard can be used to represent RETURN.

Examples:
```
F7=%  .BYTE  %
F4=ASM,L
```


4.3 ENTERING LINES OF ASSEMBLY LANGUAGE

If you type in a line number followed by a line of 6502 assembly language, the editor will put the line into memory according to its line number (these may be 1 to 65535). A line number followed by RETURN deletes that line, and a line with number zero will be put immediately after the last line entered or deleted.

If the editor finds a syntax error in a line of source code, it prints an arrow pointing to the error and an error message. This feature can be suppressed using the EDITOR command.
In any situation which could result in the destruction of the source text, the editor will prompt with 'ARE YOU SURE (Y/N)? ' before proceeding.

In the following list of commands, each one is followed by its abbreviated version in brackets.

4.4 EDITOR (ED.) command

This command puts the assembler into 'EDITOR' mode which disables the automatic syntax checking of lines; in this mode the text is not tokenised and therefore cannot be assembled. Entering or leaving this mode destroys any text that is in memory.

4.5 RESIDENT (RES.) command

This command puts the assembler into 'RESIDENT' mode in which programs may be assembled directly from memory. Further details are given in section 9 of this manual.

4.6 DISK (DISC or DI.) command

This command puts the assembler into 'DISK' mode in which programs may be assembled from disk. Further details are given in section 10 of this manual.

4.7 LIST (L.) command

This is used to list lines of text. It may be followed by one or more line specifications (separated by semicolons) of the following types:

```
<line number>
<first line>,<last line>
,<last line>
<first line>,
```

Missing out the line specifications will list the whole source text.

Examples:
```
LIST
LIST 23790
L. !PRINTMNEMONIC,
LIST ,100;110;150,160
L. 23990
LIST 100,100+70
```

4.8 PRINT (P.) command

This is the same as LIST but no line numbers are printed.

4.9 DELETE (D.) command

This command is used to delete lines from the source text- the syntax is the same as for LIST. The editor will list the lines and then prompt with 'ARE YOU SURE (Y/N)? ' before the lines are actually deleted - hit 'Y' to carry out the deletion.

Examples:
```
D. 23770
DEL 1440;2680;3725,3737
```

4.10 RENUMBER (R.) command

This renumbers lines in the text. It may take any of the following forms:

RENUMBER
first line no.=10, step size10

RENUMBER X
first line no.=X, step size=10

RENUMBER X,Y
first line no.=X, step size=Y

If renumbering would cause a line number greater than 65535 to be generated, the text is renumbered from line 1 in steps of 1. After renumbering, the last line number + step size is printed.

Examples:
```
RENUMBER
R.10000
REN. 100,25
```

4.11 MEM (M.) command

This command returns a message giving the free memory total and the current editor mode. If the source file is very long there will be a delay of a few seconds while symbol table garbage collection is carried out.

4.12 NEW (N.) command

This erases the source program in memory and then prompts for the memory to be reserved for source text.

4.13 AUTO (AU.) command

This puts the computer into AUTO mode- after a line number and text is entered, the next line number is automatically printed. The default value of the step size is 10. This can be changed by placing the new value after the command. If the step size is zero, line number 0s only are printed. To stop the printing of the numbers, enter a blank line.

Examples:
```
AU.
AUTO 5
```

4.14 MANUAL (MA.) command

This brings the computer out of AUTO mode.

4.15 MOVE (MO.) command

This command is used to move a block of lines from one part of the text to another. It takes the following form:

MOVE <new line number><first line>,< last line>

After the lines have been moved, the first line is renumbered to the <new line number>, the rest being renumbered to line 0. If the <new line number> already exists, an error message is printed.

Examples:
```
MOVE 1475=2510,2850
MO. 23000=570,810
```

4.16 COPY (CO.) command

This is similar to MOVE, the difference being that the lines are not deleted from their original position once they have been moved

4.17 FIND (FI.) command

This command is used to find the location of a sequence of characters in the text. It takes the following forms:

FIND <string>
FIND <string> <line specification>

The second form should be used if it is desired to search only a part of the text. Wher the <string> is found, the line in which it appears is printed.

Examples:
```
FIND "JSR"
FIND & HEXOUT& 1140,2930
```

4.18 CHANGE (CH.) command

This command is used to change all occurrences of a particular series of characters. It should be followed by two strings and a line specification. The delimiter at the end of the first string should not be duplicated at the start of the second string.

Examples:
```
CH.!HEXOUT!HEX$BYTE$OUT!
CHANGE %JSR OUTCH%JSR OUTCHR% 3980,7620
```

5. EDITOR ERROR MESSAGES

The following error messages can be generated by the editor:

OUT OF FUNCTION KEY SPACE
All the function key definitions may not total more than 119 chracters.

NUMBER TOO BIG

EXPRESSION TOO COMPLEX
An expression has too many levels of nested parenthesis.

DIVISION BY ZERO

LABEL TOO LONG
A label was found which is longer than 15 characters; the editor uses only the first 15.

LABEL DOES NOT BEGIN WITH A LETTER

"A" IS A RESERVED LABEL

6502 OP-CODES ARE RESERVED LABELS
A 6502 op-code mnemonic was used as part of an expression.

BAD INDEX
Index must be X or Y. This message is also generated if either (<expression>,Y) or (<expression>),X are encountered.

BAD DIRECTIVE
A string was found after a full stop which is not one of the legal directives.

* FULL *
You have run out of memory space.

FILE ERROR

STRING TOO LONG
The maximum length allowable is 64 characters.

OUT OF RANGE
An attempt was made to move a block of text to a location within itself.

SYNTAX ERROR
The error does not fall into one of the above categories.

6. LOADING AND SAVING

6.1 LOAD (LO.) command

This is used to load source files into memory. It should be followed by a <string> for the filename.

To merge a file onto the program in memory, follow the filename with a comma (optional) and the line number where the text is to be inserted. The text is always inserted before the line specified if it exists. Using this facility, subroutines that have been previously written, debugged and saved can be incorporated into a program.

If a line is loaded which is not valid, it is listed on the screen to be corrected after loading.

Examples:
```
LOAD  XBASIC
LOAD  %HEXPRINT%  200
LO.  !A1!
LOAD
LOAD  (SOURCE(,300
```

6.2 SAVE (SA.) command

This saves the source text to disk or tape. The filename may be followed by a line specification (as defined in Section 4.7) if only part of the source file is to be saved.

Examples:
```
SAVE  $PART6$  100,400
SAVE
SAVE  @0:ASM1
```

Source text is saved in a compressed format: all unnecessary spaces are removed, and any that remain are removed and bit 7 of the next character is set.

6.3 FSAVE (F.) command

This command is similar to SAVE, but the text is saved formatted as it would be printed, without any text compression.

6.4 MLOAD (ML.) command

This is used to load machine-code files that have been saved in the standard format. To load it at a different address than it was saved at, follow the filename with the new start address; this may be preceded by a comma.

Examples:
```
MLOAD LOADER
MLOAD
MLOAD  'SPRITES',$8000
```

6.5 MSAVE (MS.) command

This saves machine-code in the standard format. The command should be followed by the filename, start address and end address, all separated by commas.

Examples:
```
MSAVE %TITLE SCREEN%,$A000,$C000
MSAVE $OUTPUT PATCH$,$C000,$C180
```

6.6 OLOAD (OL.) command

This loads object code that has been saved in ASCII format; this is the format used for object code files generated by the assembler

The command should be followed by the filename.

Example:
```
OLOAD 'OBJECT'
```

Each byte of data to be stored is converted into two half bytes which are translated into their ASCII equivalents ('0' to 'F'). Each output record begins with a ';' character. The next byte is the number of data bytes contained in the record. The record's starting address High (1 byte, 2 characters), starting address Low (1 byte, 2 characters) and data (maximum 24 bytes, 48 characters) follow. Each record is terminated by the record's checksum (2 bytes, 4 characters) and a carriage return.

The last record saved has zero data bytes (indicated by ;00). The starting address field is replaced by a four digit hexadecimal number representing the total number of data records contained in the file, followed by the record's usual checksum digits.

Examples:
```
;180000FFEEDDCCBBAA00998877665544332211223344556677889900AFC
;0000010001
```

Note: A program is supplied with Laser Genius under the filename "LOADER" which loads data in ASCII format and can be run from BASIC.

6.7 OSAVE (OS.) Command

This saves object code in ASCII format, details of which are given above. The command is followed by a filename, and the start and end addresses separated by commas. More than one block of data may be saved by separating several start and end address pairs with semicolons.

Examples:
```
SAVE &TESTFILE&,$4C00,$5000
SAVE $PART6$,$6000,$7000;$81F3,$8230
```

6.8 OC+ and OC- commands

After executing the OC+ command, all object code is saved in a compact format in which bytes are output directly to disk rather than being printed in hex and the record start character is ':' rather than ';'.

The OC- command is used to revert to the normal format.

6.9 The 'LOADER' Program

Supplied on both the tape and disc version of Laser Genius is the LOADER program which enables object code produced by the assembler's ASM,0 command to be loaded into the Commodore's memory outside of the assembler. The program works in the same way as the OLOAD command.

TAPE: The program can be found on tape 2, side A, after the turbo version of GENASM. To load use SHIFT and RUN/STOP.

DISC: The program can be loaded from disc using:

```
LOAD "LOADER",8,1
```

The program will run automatically.

Once the program has loaded it will ask for a hexadecimal offset. This offset is the same as that given by the OFFSET command (see 9.2) from within the assembler. It allows code to be loaded into memory at a different place from where it was assembled. Normally you will only need to press enter for this prompt. Next the program asks for the filename of the object code to be loaded (on the tape version you can just press enter).

Assuming that the file loads and is converted correctly the code can then be executed from BASIC using SYS or from the Monitor/Analyser using slow running or single stepping.

7. USING A PRINTER

7.1 CENTRO (C.) command

The assembler is set to use a serial bus printer (device number 4) upon initialisation. To use it with a centronics interface printer connected to the user port, type 'CENTRO+'. To revert to the serial bus printer, use 'CENTRO-'.

7.2 CTRL (CT.) command

This can be used to send a series of control codes to the printer for initialisation purposes. It should be followed by one or more <expression>s separated by commas.

Example:
```
CTRL 27,'M
```

7.3 * command

Placing an asterisk before any command will direct its output to the printer.

Examples:
```
*L.4920,5260
*ASM,L,C
```

7.4 Printer Pagination

At the top of each new page, a heading consisting of a title and a page number is printed. The following commands are available (some are also assembler directives):

7.4.1 INTNUM command

This sets the current page number to zero.

7.4.2 SETPAGE command

This is used to define the paper size. The command can exist in either of the following forms:

```
SETPAGE X    Sets the paper width to X.
SETPAGE X,Y  Sets the paper width to X and page height to Y.
```

The minimum value for either of these parameters is 16; the maximum is 127. To disable paging, set the page length to zero.

7.4.3 SKIP command.

This is used to skip a certain number of lines.

Example:
```
SKIP 15
```

7.4.4 TITLE command

This sets the title that is printed at the top of each new page.

Example:
```
TITLE SOURCE CODE LISTING
```

7.5 Setting up the Printer.

After loading the assembler, position the print head at the top of a new page. This ensures that subsequent page headings will be properly aligned.

The paper width and height are set to 80 and 66 respectively after 64-MAC/MON has been loaded.

7.6 P# Command

This command is used to change the device number which 64-MAC/MON uses for the printer. Normally this is set to device number 4. To change it, follow P# by the new device number.

Example: To change the device number to 6, for a printer/plotter, you would use:

```
P# 6
```


8. DOS SUPPORT

8.1 @ (or >) command

This sends a command to the disk drive. The legal commands are:

8.1.1 Format a Disk: @N<drive number>:<disk name>,XX

XX is a unique 2-character identifier; omitting it, results in all files being deleted rather than re-formatting the whole disk.

Example:
```
@N0:DISK 1,99
```

8.1.2 Delete a File: @S<drive number>:<filename>

Pattern matching using '*' and '?' may be used to delete groups of files.

Examples:
```
>S1:ASM*
@S0:OBJECT
```

8.1.3 Rename a File: @R<drive number>:<new file name>=<old file name>

Example:
```
@R0:PROGRAM=P6
```

8.1.4 Validate a Disk: @V<drive number>

This reconstructs the Block Availability Map on the disk. If you suspect that a disk is corrupted, this command will prevent further corruption of files. It should also be used if you have any files on the disk that are not properly closed.

8.1.5 Duplicate Disk: @D<destination drive number>=<source drive number>

Example:
```
@D1=0
```

8.1.6 Copy File: @C<drive number>:<new file>=<drive number>:<old file>

This command can also be used to concatenate several files:

@C<drive number>:<new file>=<drive number>:<file 1>,<drive number>:<file 2>

A maximum of four files can be joined in this manner.

Examples:
```
>C0:PROG2=0:PROG1
@C1:SOURCE=0:A1,0:A2
```

8.1.7 Print Directory: @$<drive number>:<filename>

The 'DIR' command can be used instead of '@$'

Examples:
```
>$
@$0:ASM*
@$1:MONITOR$?000
DIR  $0:A*
DIR
```

8.1.8 Read Error Channel: @ or > or ERR

This will print out an error number, error name, and track and sector numbers.

Further details of these commands can be found in your disk drive manual. For a single disk drive, the <drive number> should always be 0.

8.2 Pattern Matching

Pattern matching can be used with LOAD and DOS commands. The two symbols used are '*' (signifies 'with anything following') and '?', which matches with any character.

For example, 'COM????R*' can be used to specify 'COMMODORE' and 'COMPUTERS' but not 'COMBINATION'.

8.3 DEVICE (DEV. or #) command

This is used to change the device number used in LOAD, SAVE and DOS commands.

Example: 'DEVICE 1' will allow loading and saving of files from tape.

8.4 Using more than one Disk Drive Unit.

Drive numbers 2 to 7 may be used to specify device numbers 9 to 11 as shown in the table below:

USER FILENAME	DEVICE	DISK FILENAME
0:filename	8	0:filename
@0:filename	8	@0:filename
1:filename	8	1:filename
@1:filename	8	@1:filename
2:filename	9	0:filename
@2:filename	9	@0:filename
3:filename	9	1:filename
@3:filename	9	@1:filename
4:filename	10	0:filename
@4:filename	10	@0:filename
5:filename	10	1:filename
@5:filename	10	@1:filename
6:filename	11	0:filename
@6:filename	11	@0:filename
7:filename	11	1:filename
@7:filename	11	@1:filename

N.B. Due to the lack of bus arbitration on the serial bus, the system will hang if you try to write to one disk drive while reading from another.

8.5 The '↑' Filename

If you save a file under the filename '↑', the assembler will use the filename in a comment on the first line of the source program currently in memory. For example, if the first line is:

```
10    ;@0:PROG
```

typing 'SAVE ↑' is equivalent to typing 'SAVE @0:PROG'.

9. THE ASSEMBLER IN RESIDENT MODE

In RESIDENT mode, the source file is held in memory.

9.1 ASM (A.) command

This command assembles the source file. It may be followed by letters, preceded by commas, which are used to select various options:

L	A full assembler listing is generated.
M	Assembles directly to memory.
O	Assembles object code to tape or disk.
C	A concordance listing is generated.

Examples:
```
ASM,O
A.,L,C
A.
```


9.1.1 Each line of the assembler listing consists of the following:

line number, address (in hex), object code, source line.

9.1.2 If object code is assembled to tape or disk, you are prompted for the object filename before assembly starts. The object code file is closed if any errors occur. Note that the object code generated by the assembler cannot be loaded directly into memory - it must be loaded using either the OLOAD command in the 64 MAC/MON or the special object code loader program supplied.

9.1.3 Two concordance listings are actually printed: the first has the labels in alphabetical order, and the second has them in numerical order. Each line of the concordance listing contains the label, its value, the line that it was defined in and the lines in which it was referred to. This is printed after the assembler listing.

9.1.4 If an error is found, a pointer will be printed pointing to the error, followed by an error message. A full list of error messages is given in Part 11 of this manual.

9.2 OFFSET (O.) command

This sets an offset which is added to the location counter when object code is being output using the O or M options. This offset also applies to the OLOAD, OSAVE and monitor-type commands.

Examples:
```
OFFSET $9800
O.$E000
```

9.3 RUN command

This command assembles and executes an assembly language program, allocating memory to place the object code in.
The start address of the program itself should be defined with a .ORG directive (see Section 2.2.1.9), but when the location counter is set to reserve space, *= should be used.

10. THE ASSEMBLER IN DISK MODE

10.1 Linked Files

In DISK mode, the source program is held on disk as a series of linked files. The files are linked with .FILE directives (see Section 2.2.1.30). The last file should finish with an .END directive (Section 2.2.1.5).

10.2 Using Macros in Disk Mode

All macros must be defined in the first file; this is kept in memory throughout the assembly. This file should be short, so as to leave as much space as possible for the symbol table. If there is a symbol table overflow, a '* FULL *' error message is printed, and assembly is aborted.

10.3 ASM (A.) command

In disk mode, the ASM command should not be followed by option letters as in resident mode. Instead, the computer asks if a listing, concordance listing or object code is to be generated. When prompted for the source filename, give the name of the first file.

If there is a source file in memory, you are given the option of saving it since it will be destroyed by assembly in disk mode; if you do not wish to save it, hit return when you are prompted for the filename.

11. ASSEMBLER ERRDR MESSAGES

The assembler can generate the following error messages:

FWD REF OR UNDEFINED LABEL IN .BLOCK OR ORIGIN DIRECTIVE

.BYTE DIRECTIVE DATA TOO BIG
The .BYTE directive can only accept data less than 256.

BRANCH OUT OF RANGE
A relative branch must be to an address within the range * 126 to *+129.

BAD OP-CODE/OPERAND COMBINATION

LABEL ALREADY DEFINED

IMMEDIATE OPERAND TOO BIG
Immediate operands must be less than 256.

IRRESOLVABLE FWD REF OR UNDEFINED LABEL
The label has not been defined in the source file.

MACROS NESTED TOO DEEP
Macros can be nested up to a maximum of 32 deep.

TOO MANY .ENDMACS
An .ENDMAC was found without a corresponding .DEFMAC.

TOO MANY .DEFMACS
A .DEFMAC was found inside a macro definition. This message is also generated if the assembler runs off the end of the source text in the middle of a macro.

WRONG NUMBER OF PARAMETERS
The wrong number of parameters were used in a macro call.

UNDEFINED MACRO

CONCORDANCE TABLE OVERFLOW
This message is printed at the end of assembly.

CONTEXT ERROR IN DISK FILE
Disk mode only; a line was found in the source file which is not a valid line of 6502 assembly language. This probably means that the disk is corrupted.

ASSEMBLY TO RESERVED MEMORY
An attempt was made to assemble on top of zero page, the source text, the symbol table or the assembler. This also happens if there is not enough room for the object code in the RUN command. This message is only given if an assembly to memory is being carried out.

12. MACHINE CODE MONITOR COMMANDS

These commands allow you to inspect and modify memory directly; they include a symbolic disassembler. The offset as set by the OFFSET command is added to all locations when they are referred to.

A <byte string> is defined as a series of <string constant>s (see Section 2.1.4) and/or expressions (see Section 3.1), separated by commas.

Example: `1,2,'ABC',9`

12.1 DECIMAL (DE.) command

This puts the monitor commands into DECIMAL mode: all numerical output is in base 10.

12.2 HEX (H.) command

This puts the monitor commands into HEX mode: All numerical output is in base 16.

12.3 CALC (?) command

This evaluates an expression and prints the result as an ASCII character, and in binary, octal, decimal and hex.

Examples:
```
CALC 'A'+1
'B' %01000010 @102 066 $42
READY.

? (9+3)/4
'.' %00000011 @003 003 $03
READY.
```

12.4 MLIST command

This prints the contents of memory in both numerical and ASCII form. It can take two forms:

MLIST <start address>
Lists memory 8 lines at a time. Hit <return> to continue, otherwise type in the next command.

MLIST <start address>,<end address>
Lists a block of memory continuously.

For a different output format, replace MLIST in the above with:

MLIST@ <no. bytes per line>,<no. spaces between bytes>,<no. linefeeds between lines>,

To change a byte in memory, simply move the cursor over it, change it and hit <return>. If you don't hit <return>, the byte in memory will not be changed in memory even if it is changed on the screen.

12.5 MDUMP (DUMP or MD.) command

This is similar to MLIST, the difference being that memory is not printed as ASCII characters.

12.6 MFIND (MF.) command

This prints the addresses of all occurrences of a sequence of bytes between two addresses:

MFIND <byte string> > <start address>,<end address>

Examples:
```
MFIND $68,$20,$15,$DF>$E061,$F83C
MFIND 'BASIC'>$A000,$C000
```

12.7 COMPARE (CD.) command

This compares one block of memory with another, printing any differences:

COMPARE <start address 1>=<start address 2>,<end address 2>

Example: If the memory contents are:

```
$8000: $01 $02 $03 $04 $05 $06 $07 $08
$9000: $01 $03 $02 $04 $05 $66 $07 $08
```

then the following would be printed:

```
COMPARE $8000=$9000,$9008

$8001=$02,$9001=$03
$8002=$03,$9002=$02
$8005=$06,$9005=$66
READY.
```

12.8 MFILL (FILL) command

This fills a block of memory with a sequence of one or more bytes:

MFILL <start address>,<end address><byte string>

Examples:
```
MFILL $D800,$DC00=0,1,2,3,4,5,6,7,8,9,
                  10,11,12,13,14,15
MFILL $8000,$9000=$EA
```

12.9 MMOVE (MM.) command

This moves a block of memory from one place to another:

MMOVE <new start address> <old start address>,<old end address>

Example:
```
MMOVE $B000=$9012,$9200
```

12.10 RELOC (REL.) command

This is similar to MMOVE, the difference being that all JMPs etc. are changed so that the program will run at the new address.

Example:
```
RELOC $5000=$A000,$C000
```

It is also possible to place the relocated program in a different part of memory from where it will run:

RELOC <memory address>,<run address>=<old block start>,<old block end>

12.11 MCHANGE (MCH.) command

This finds all occurrences of a sequence of bytes and replaces it with another:

MCHANGE <old byte string> > <new byte string> > <start address>,<end address>

Example:
```
MCHANGE$20,$91,$19>$20,$95,$19>$C900,$CA00
```

12.12 USR (U.) command

This performs a user-defined operation on a block of memory. A subroutine must be supplied which carries out the required operation on the accumulator:

USR <subroutine address>=<start of block>,<end of block>

Example: Add 1 to all locations between $8000 and $9000

1. Assemble into memory:

```
10 *=$9800
20          CLC
30          ADC  #1
40          RTS
```

2.
```
USR  $9800=$8000,$9000
```

The X and Y registers may be used in the subroutine

12.13 DUSR (DU.) command

This carries out a user-defined operation on double-byte quantities in a block of memory. A subroutine must be supplied which carries out the required operation on the accumulator (LOW) and X (HIGH).

Example: Subtract 1 from each entry in a table of addresses between $CB00 and $CB4C.

Assemble into memory:

```
10 *=$C000
20          SEC
30          SBC #1
40          BCS RETURN
50          DEX
60 RETURN RTS
```

2.
```
DUSR $C000=$CB00,$CB4C
```

The Y register may be used in the subroutine.

12.14 The Symbolic Disassembler

12.14.1 BYTE (B.) command

This is used to define blocks of memory which are printed as .BYTE directives when disassembling; the command should be followed by the lower and upper limits of the block, separated by commas. Up to 16 blocks can be held in memory at once, and more than one can be defined at a time by separating the blocks by semicolons.

Example:
```
BYTE  $9164,$9197;$9304,$9308
```

12.14.2 ASCII (ASC.) command

This is similar to .BYTE, but any ASCII characters are output as <string constant>s.

12.14.3 WORD (W.) command

This is similar to .BYTE, but it defines blocks which are printed as .WORD directives.

12.14.4 ØBYTE (DB.) command

This is similar to .BYTE, but it defines blocks which are printed as .DBYTE directives.

12.14.5 TABLES (TA.) command

This prints out a table of the blocks of memory defined in the above four commands.

12.14.6 TABDEL command

This has the same syntax as the byte command, but it deletes an entry from the table.

12.14.7 TABCLR command

This removes all entries from the table defined by the BYTE, ASCII, WORD and DBYTE commands.

12.14.8 Defining Symbols for use by the Disassembler

If the symbol table has been destroyed by printing out the 'BYTES FREE' message, the disassembler output is non-symbolic, and the address and object code are also printed. After an error-free assembly, the disassembler will search the symbol table when printing out any constants. The address and object code is not printed out in this mode. Thus, an assembler source file consisting of '=' directives should be used to define symbols for use by the disassembler; the number of symbols is only limited by the memory size, about 1500 if the full 34k is being used for text storage.

12.14.9 DASM (DA.) command

This command is used to disassemble machine code in memory. It can take two forms:-

DASM <address>
This disassembles in blocks of 23 lines. Hit <return> to continue, or type in the next command.

DASM <start address>,<end address>
This disassembles a block of memory continuously.

12.14.10 FDASM command

This is used to disassemble to disk or tape:

FDASM <filename> <start address>,<end address>

Example:
```
FDASM  #BASIC#  $A000,$C000
```

12.15 SYS command

This command calls a machine code subroutine.

Example: SYS $9804

12.16 Although all of the available zero page is used by the 64 MAC/MON, this memory can be used in your own routines as it is exchanged with a set of temporary storage locations when fetching or storing bytes in memory. Also, in the high memory version, the BASIC ROM is switched on when accessing memory, so that it is possible to disassemble it or use subroutines from it in your own programs.

13. MONITOR ERROR MESSAGES

The monitor commands can generate the following error messages:

WRONG LENGTH
The two byte strings in a MCHANGE command are of different length.

NOT IN TABLE
An attempt was made to delete a non existant table entry.

TABLE FULL
The tables can each hold only 16 entries.

OUT OF RANGE
An attempt was made to RELOCate a program to a location within itself.

14. THE DEBUGGER/TRACER

14.1 OPT command

This command is used to select the tracer's mode of operation. The command should be followed by zero or more of the following letters, each preceded by a comma:

J : JSR mode.
S : STEP mode.
A : ADDRESS mode.
R : REGISTER mode.

14.1.1 If JSR mode is enabled, all JSR instructions are executed rather than traced, and tracing stops when a RTS instruction is found. This mode should be used for debugging subroutines, where the lower level subroutines have already been similarly debugged.

14.1.2 In STEP mode, you must press return each time the registers are displayed. To terminate the trace, type in the next command as usual.

14.1.3 In ADDRESS mode, the address of each instruction executed is printed out.

14.1.4 In REGISTER mode, the register contents are printed at each instruction. If this mode is disabled, they are only displayed upon 'display-points' (see section 14.2).

14.2 DISP command

This is used to set the 'display points' at which the register contents are always printed out. Several display points may be specified by separating them with commas. A maximum of 16 display points may be specified. 'Display points' are not the same as conventional breakpoints ; they are detected by the software in the tracer rather than

using a hardware BRK instruction, and they therefore are not detected if a machine code program is run using the SYS command, but, unlike breakpoints, they may be set in ROM routines.

Example: `DISP $8000,$8120`

14.3 DISTAB command

This prints out a table of all the current 'display points'.

14.4 DISDEL command

This has the same syntax as DISP but it deletes items from the table.

14.5 DISCLR command

This clears all entries from the display point table.

14.6 REGS command

This prints out the contents of the CPU registers as used by TRACE. The register contents can be modified in the same way as memory locations with MLIST (see section 12.4). Only the contents of A, X, Y, S and P are relevant to the SYS instruction - SYS ignores the value of PC as given by REGS and the value of PC after SYS is invalid.

14.7 LOC command

This has the same syntax as the DISP command and is used to specify memory locations whose contents are always printed out along with the registers.

14.8 LOCDEL command

This has the same syntax as LOC but it deletes items from the table.

14.9 LOCCLR command

This deletes all entries from the table of locations specified by the LOC command. (A LOCTAB command is not included since the REGS command will print out a list of the locations in the table).

14.10 TRACE command

This starts tracing at the memory location given by PC's value as it is printed out by the REGS command. If a BRK or an illegal instruction is found while tracing, a message is printed containing PC's current value and tracing is halted.

15. 6502 ARCHITECTURE

15.1 Byte-length registers:

A (accumulator)
P (processor status flag register)
S (stack pointer)
X (index register X)
Y (index register Y)

The general purpose user registers are A, X and Y. The stack pointer always contains the least significant byte of the next available stack location in page 1 ($0100 to $01FF). The P register consists of a set of seven status flags.

15.2 Word-length registers:

PC (computer counter)

Note: Pairs of memory locations in page zero may be used as word-length registers to hold indirect addresses. The lower address holds the least significent (or low) byte and the higher holds the most significant (or high) byte. Since the 6502 provides automatic wraparound, addresses $00FF and $0000 provide a rarely used pair.

15.3 Flags:

The flags are arranged in the P register as follows:

bit	flag	purpose
0	C	Carry
1	Z	Zero
2	I	IRQ interrupt disable
3	D	Decimal mode
4	B	BRK command
5	X	Unused (always set to 1)
6	V	Overflow
7	N	Negative (sign)

16. THE 6502 INSTRUCTION SET

ADC - Add memory to accumulator with carry.
flags affected: N,Z,C,V (Z is invalid if in decimal mode).

AND - Logical 'and' accumulator with memory.
flags affected: N,Z

ASL - Arithmetic shift left.
(Bit 7 goes to C flag, a 0 is shifted into bit zero). flags affected: N,Z,C

BCC - Branch to destination if C flag=0

BCS - Branch to destination if C flag=1

BEQ - Branch to destination if Z flag=1

BIT - Bit test. Logical 'and's ACC with memory and sets Z on the result but does not alter the contents of ACC. Bit 7 of memory goes to the N flag, and bit 6 goes to the V flag.
flags affected: N,Z,V

BMI - Branch to destination if N flag=1

BNE - Branch to destination if Z flag=0

BPL - Branch to destination if N flag=0

BRK - Force IRQ interrupt.

BVC - Branch to destination if V flag=0

BVS - Branch to destination if V flag=1

CLC - Clear the carry flag.
flags affected: C(=0)

CLD - Clear the decimal mode flag.
flags affected: D(=0)

CLI - Clear the interrupt disable flag.
flags affected: I(=0)

CLV - Clear the overflow flag.
flags affected: V(=0)

CMP - Compare accumulator with memory.
flags affected: N,Z,C

CPX - Compare X index register with memory.
flags affected: N,Z,C

CPY - Compare Y index register with memory.
flags affected: N,Z,C

DEC - Decrement memory.
flags affected: N,Z

DEX - Decrement X index register.
flags affected: N,Z

DEY - Decrement Y index register.
flags affected: N,Z

EOR - Exclusive-or memory with accumulator.
flags affected: N,Z

INC - Increment memory.
flags affected: N,Z

INX - Increment X index register.
flags affected: N,Z

INY - Increment Y index register.
flags affected: N,Z

JMP - Jump to new location.

JSR - Jump to a subroutine.Pushes the program counter+2 onto the stack and then jumps to the location.

LDA - Load accumulator from memory.
flags affected: N,Z

LDX - Load X index register from memory.
flags affected: N,Z

LDY - Load Y index register from memory.
flags affected: N,Z

LSR - Logical shift right. (Bit zero goes to carry, and a zero is shifted into bit 7.)
flags affected: N(=0),Z

NOP - No operation.

ORA - Logical 'or' accumulator with memory.
flags affected: N,Z

PHA - Push accumulator onto the stack.

PHP - Push processor status register onto the stack.

PLA - Pull accumulator from the stack.
flags affected: N,Z

PLP - Pull procesor status register from the stack.
flags affected: restored

ROL - Rotate left through carry. (Carry is shifted into bit 0 and bit 7 is shifted into carry.)
flags affected: N,Z,C

ROR - Rotate right through carry. (Carry is shifted into bit 7 and bit 0 is shifted into carry.)
flags affected: N,Z,C

RTI - Return from interrupt. Pulls status register and program counter off the stack.
Flags affected: restored

RTS - Return from subroutine. Pulls an address off the stack, adds 1 and jumps to that location.

SBC - Subtracts memory from accumulator with carry. (Carry acts as an inverted borrow.)
flags affected: N,Z,C,V (Z is invalid if in decimal mode).

SEC - Set the carry flag.
flags affected: C(=1)

SED - Set the decimal mode flag.
flags affected: D(=1)

SEI - Set the interrupt disable flag.
flags affected: I(=1)

STA - Store accumulator in memory.

STX - Store the X-index register in memory.

STY - Store the Y-index register in memory.

TAX - Transfer the accumulator to the X index register.
flags affected: N,Z

TAY - Transfer the accumulator to the Y index register.
flags affected: N,Z

TSX - Transfer the stack pointer to the X index register.
flags affected: N,Z

TXA - Transfer the X index register to the accumulator.
flags affected: N,Z

TXS - Transfer the X index register to the stack pointer.

TYA - Transfer the Y index register to the accumulator.
flags affected: N,Z

17. THE 6502 ADDRESSING MODES

N.B. All 16-bit addresses are stored in memory with the least significant byte first.

17.1 IMMEDIATE ADDRESSING

The operand is contained in the second byte of the instruction.

Length: 2 bytes
Assembler Notation: #<expression>
Example: LDA #CR

17.2 ABSOLUTE ADDRESSING

The 2nd and 3rd bytes of the instruction form the effective address.

Length: 3 bytes
Assembler Notation: <expression>
Example: INC 34975

17.3 ABSOLUTE,X ADDRESSING

The effective address is formed by adding the X-register to the address in the 2nd and 3rd bytes of the instruction.

Length: 3 bytes
Assembler Notation: <expression>,X
Example: CMP TABLE,X

17.4 ABSOLUTE,Y ADDRESSING

The effective address is formed by adding the Y-register to the address in the 2nd and 3rd bytes of the instruction.

Length: 3 bytes
Assembler Notation: <expression>,Y
Example: INC $1000,Y

17.5 ZERO PAGE ADDRESSING

The second byte of the instruction is the low-order 8 bits of the effective address; the high-order byte is zero.

Length: 2 bytes
Assembler Notation: <expression>
Example: INC 100

17.6 ZERO PAGE,X ADDRESSING

The X register is added to the 2nd byte of the instruction to give the low-order 8 bits of the effective address; the high-order byte is always zero.

Length: 2 bytes
Assembler Notation: <expression>,X
Example: STA BUFFER,X

17.7 ZERO PAGE,Y ADDRESSING

The Y register is added to the 2nd byte of the instruction to give the low-order 8 bits of the effective address; the high-order byte is always zero.

Length: 2 bytes
Assembler Notation: <expression>,Y
Example: STX $33,Y

17.8 RELATIVE ADDRESSING

The 2nd byte of the instruction is a signed offset which is added to the program counter to give the effective address. The assembler automatically calculates the offset from the operand given.

Length: 2 bytes
Assembler Notation: <expression>
Example: BNE LOOP

17.9 ACCUMULATOR ADDRESSING

The accumulator is the operand of the instruction.

Length: 1 byte
Assembler Notation: A
Example: LSR A

17.10 IMPLIED ADDRESSING

This addressing mode is implied by the instruction; no operand exists.

Length: 1 byte
Example: DEY

7.11 INDIRECT ADDRESSING

The 2nd and 3rd bytes of the instruction contain a pointer to the 16-bit effective address of the instruction. Due to an error in the 6502's design, this will not work correctly if the 2nd and 3rd bytes of the instruction cross a page boundary.

Length: 3 bytes
Assembler Notation: (<expression>)
Example: JMP (VECTOR)

17.12 INDIRECT,Y ADDRESSING

The effective address is calculated by adding the Y register to a 16-bit address contained in page zero which is pointed to by the 2nd byte of the instruction.

Length: 2 bytes
Assembler Notation: (<expression>),Y
Example: LDA (PTR),Y

17.13 INDIRECT,X ADDRESSING

The 2nd byte of the instruction and the X register are added to give the address of two locations in page zero which hold the effective address.

Length: 2 bytes
Assembler Notation: (<expression>,X)
Example: LDA (BRKTAB,X)

The notations for zero page and absolute addressing modes are the same - the assembler decides which mode to use.

18. NUMBER BASE CONVERSION TABLE WITH 6502 OP-CODES

```
%00000000 @000 000 $00 BRK IMPLIED
%00000001 @001 001 $01 ORA INDIRECT,X
%00000010 @002 002 $02 UNUSED
%00000011 @003 003 $03 UNUSED
%00000100 @004 004 $04 UNUSED
%00000101 @005 005 $05 ORA ZERO PAGE
%00000110 @006 006 $06 ASL ZERO PAGE
%00000111 @007 007 $07 UNUSED
%00001000 @010 008 $08 PHP IMPLIED
%00001001 @011 009 $09 ORA IMMEDIATE
%00001010 @012 010 $0A ASL ACCUMULATOR
%00001011 @013 011 $0B UNUSED
%00001100 @014 012 $0C UNUSED
%00001101 @015 013 $0D ORA ABSOLUTE
%00001110 @016 014 $0E ASL ABSOLUTE
%00001111 @017 015 $0F UNUSED
%00010000 @020 016 $10 BPL RELATIVE
%00010001 @021 017 $11 ORA INDIRECT,Y
%00010010 @022 018 $12 UNUSED
%00010011 @023 019 $13 UNUSED
%00010100 @024 020 $14 UNUSED
%00010101 @025 021 $15 ORA ZERO PAGE,X
%00010110 @026 022 $16 ASL ZERO PAGE,X
%00010111 @027 023 $17 UNUSED
%00011000 @030 024 $18 CLC IMPLIED
%00011001 @031 025 $19 ORA ABSOLUTE,Y
%00011010 @032 026 $1A UNUSED
%00011011 @033 027 $1B UNUSED
%00011100 @034 028 $1C UNUSED
%00011101 @035 029 $1D ORA ABSOLUTE,X
%00011110 @036 030 $1E ASL ABSOLUTE,X
%00011111 @037 031 $1F UNUSED
%00100000 @040 032 $20 JSR ABSOLUTE
%00100001 @041 033 $21 AND INDIRECT,X
%00100010 @042 034 $22 UNUSED
%00100011 @043 035 $23 UNUSED
%00100100 @044 036 $24 BIT ZERO PAGE
%00100101 @045 037 $25 AND ZERO PAGE
%00100110 @046 038 $26 ROL ZERO PAGE
%00100111 @047 039 $27 UNUSED
%00101000 @050 040 $28 PLP IMPLIED
%00101001 @051 041 $29 AND IMMEDIATE
%00101010 @052 042 $2A ROL ACCUMULATOR
%00101011 @053 043 $2B UNUSED
%00101100 @054 044 $2C BIT ABSOLUTE
%00101101 @055 045 $2D AND ABSOLUTE
%00101110 @056 046 $2E ROL ABSOLUTE
%00101111 @057 047 $2F UNUSED
%00110000 @060 048 $30 BMI RELATIVE
%00110001 @061 049 $31 AND INDIRECT,X
%00110010 @062 050 $32 UNUSED
%00110011 @063 051 $33 UNUSED
%00110100 @064 052 $34 UNUSED
%00110101 @065 053 $35 AND ZERO PAGE,X
%00110110 @066 054 $36 ROL ZERO PAGE,X
%00110111 @067 055 $37 UNUSED
%00111000 @070 056 $38 SEC IMPLIED
%00111001 @071 057 $39 AND ABSOLUTE,Y
```

```
%00111010 @072 058 $3A UNUSED
%00111011 @073 059 $3B UNUSED
%00111100 @074 060 $3C UNUSED
%00111101 @075 061 $3D AND ABSOLUTE,X
%00111110 @076 062 $3E ROL ABSOLUTE,X
%00111111 @077 063 $3F UNUSED
%01000000 @100 064 $40 RTI IMPLIED
%01000001 @101 065 $41 EOR INDIRECT
%01000010 @102 066 $42 UNUSED
%01000011 @103 067 $43 UNUSED
%01000100 @104 068 $44 UNUSED
%01000101 @105 069 $45 EOR ZERO PAGE
%01000110 @106 070 $46 LSR ZERO PAGE
%01000111 @107 071 $47 UNUSED
%01001000 @110 072 $48 PHA IMPLIED
%01001001 @111 073 $49 EOR IMMEDIATE
%01001010 @112 074 $4A LSR ACCUMULATOR
%01001011 @113 075 $4B UNUSED
%01001100 @114 076 $4C JMP ABSOLUTE
%01001101 @115 077 $4D EOR ABSOLUTE
%01001110 @116 078 $4E LSR ABSOLUTE
%01001111 @117 079 $4F UNUSED
%01010000 @120 080 $50 BVC RELATIVE
%01010001 @121 081 $51 EOR INDIRECT,Y
%01010010 @122 082 $52 UNUSED
%01010011 @123 083 $53 UNUSED
%01010100 @124 084 $54 UNUSED
%01010101 @125 085 $55 EOR ZERO PAGE,X
%01010110 @126 086 $56 LSR ZERO PAGE,X
%01010111 @127 087 $57 UNUSED
%01011000 @130 088 $58 CLI IMPLIED
%01011001 @131 089 $59 EOR ABSOLUTE,Y
%01011010 @132 090 $5A UNUSED
%01011011 @133 091 $5B UNUSED
%01011100 @134 092 $5C UNUSED
%01011101 @135 093 $5D EOR ABSOLUTE,X
%01011110 @136 094 $5E LSR ABSOLUTE,X
%01011111 @137 095 $5F UNUSED
%01100000 @140 096 $60 RTS IMPLIED
%01100001 @141 097 $61 ADC INDIRECT,X
%01100010 @142 098 $62 UNUSED
%01100011 @143 099 $63 UNUSED
%01100100 @144 100 $64 UNUSED
%01100101 @145 101 $65 ADC ZERO PAGE
%01100110 @146 102 $66 ROR ZERO PAGE
%01100111 @147 103 $67 UNUSED
%01101000 @150 104 $68 PLA IMPLIED
%01101001 @151 105 $69 ADC IMMEDIATE
%01101010 @152 106 $6A ROR ACCUMULATOR
%01101011 @153 107 $6B UNUSED
%01101100 @154 108 $6C JMP INDIRECT
%01101101 @155 109 $6D ADC ABSOLUTE
%01101110 @156 110 $6E ROR ABSOLUTE
%01101111 @157 111 $6F UNUSED
%01110000 @160 112 $70 BVS RELATIVE
%01110001 @161 113 $71 ADC INDIRECT,Y
%01110010 @162 114 $72 UNUSED
%01110011 @163 115 $73 UNUSED
%01110100 @164 116 $74 UNUSED
%01110101 @165 117 $75 ADC ZERO PAGE,X
%01110110 @166 118 $76 ROR ZERO PAGE,X
```



```
%01110111 @167 119 $77 UNUSED
%01111000 @170 120 $78 SEI IMPLIED
%01111001 @171 121 $79 ADC ABSOLUTE,Y
%01111010 @172 122 $7A UNUSED
%01111011 @173 123 $7B UNUSED
%01111100 @174 124 $7C UNUSED
%01111101 @175 125 $7D ADC ABSOLUTE,X
%01111110 @176 126 $7E ROR ABSOLUTE,X
%01111111 @177 127 $7F UNUSED
%10000000 @200 128 $80 UNUSED
%10000001 @201 129 $81 STA INDIRECT,X
%10000010 @202 130 $82 UNUSED
%10000011 @203 131 $83 UNUSED
%10000100 @204 132 $84 STY ZERO PAGE
%10000101 @205 133 $85 STA ZERO PAGE
%10000110 @206 134 $86 STX ZERO PAGE
%10000111 @207 135 $87 UNUSED
%10001000 @210 136 $88 DEY IMPLIED
%10001001 @211 137 $89 UNUSED
%10001010 @212 138 $8A TXA IMPLIED
%10001011 @213 139 $8B UNUSED
%10001100 @214 140 $8C STY ABSOLUTE
%10001101 @215 141 $8D STA ABSOLUTE
%10001110 @216 142 $8E STX ABSOLUTE
%10001111 @217 143 $8F UNUSED
%10010000 @220 144 $90 BCC RELATIVE
%10010001 @221 145 $91 STA INDIRECT,Y
%10010010 @222 146 $92 UNUSED
%10010011 @223 147 $93 UNUSED
%10010100 @224 148 $94 STY ZERO PAGE,X
%10010101 @225 149 $95 STA ZERO PAGE,X
%10010110 @226 150 $96 STX ZERO PAGE,Y
%10010111 @227 151 $97 UNUSED
%10011000 @230 152 $98 TYA IMPLIED
%10011001 @231 153 $99 STA ABSOLUTE,Y
%10011010 @232 154 $9A TXS IMPLIED
%10011011 @233 155 $9B STA ABSOLUTE,X
%10011100 @234 156 $9C UNUSED
%10011101 @235 157 $9D UNUSED
%10011110 @236 158 $9E UNUSED
%10011111 @237 159 $9F UNUSED
%10100000 @240 160 $A0 LDY IMMEDIATE
%10100001 @241 161 $A1 LDA INDIRECT,X
%10100010 @242 162 $A2 LDX IMMEDIATE
%10100011 @243 163 $A3 UNUSED
%10100100 @244 164 $A4 LDY ZERO PAGE
%10100101 @245 165 $A5 LDA ZERO PAGE
%10100110 @246 166 $A6 LDX ZERO PAGE
%10100111 @247 167 $A7 UNUSED
%10101000 @250 168 $A8 TAY IMPLIED
%10101001 @251 169 $A9 LDA IMMEDIATE
%10101010 @252 170 $AA TAX IMPLIED
%10101011 @253 171 $AB UNUSED
%10101100 @254 172 $AC LDY ABSOLUTE
%10101101 @255 173 $AD LDA ABSOLUTE
%10101110 @256 174 $AE LDX ABSOLUTE
%10101111 @257 175 $AF UNUSED
%10110000 @260 176 $B0 BCS RELATIVE
%10110001 @261 177 $B1 LDA INDIRECT,Y
%10110010 @262 178 $B2 UNUSED
%10110011 @263 179 $B3 UNUSED
```

```
%10110100 @264 180 $B4 LDY ZERO PAGE,X
%10110101 @265 181 $B5 LDA ZERO PAGE,X
%10110110 @266 182 $B6 LDX ZERO PAGE,Y
%10110111 @267 183 $B7 UNUSED
%10111000 @270 184 $B8 CLV IMPLIED
%10111001 @271 185 $B9 LDA ABSOLUTE,Y
%10111010 @272 186 $BA TSX IMPLIED
%10111011 @273 187 $BB UNUSED
%10111100 @274 188 $BC LDY ABSOLUTE,X
%10111101 @275 189 $BD LDA ABSOLUTE,X
%10111110 @276 190 $BE LDX ABSOLUTE,Y
%10111111 @277 191 $BF UNUSED
%11000000 @300 192 $C0 CPY IMMEDIATE
%11000001 @301 193 $C1 CMP INDIRECT,X
%11000010 @302 194 $C2 UNUSED
%11000011 @303 195 $C3 UNUSED
%11000100 @304 196 $C4 CPY ZERO PAGE
%11000101 @305 197 $C5 CMP ZERO PAGE
%11000110 @306 198 $C6 DEC ZERO PAGE
%11000111 @307 199 $C7 UNUSED
%11001000 @310 200 $C8 INY IMPLIED
%11001001 @311 201 $C9 CMP IMMEDIATE
%11001010 @312 202 $CA DEX IMPLIED
%11001011 @313 203 $CB UNUSED
%11001100 @314 204 $CC CPY ABSOLUTE
%11001101 @315 205 $CD CMP ABSOLUTE
%11001110 @316 206 $CE DEC ABSOLUTE
%11001111 @317 207 $CF UNUSED
%11010000 @320 208 $D0 BNE RELATIVE
%11010001 @321 209 $D1 CMP INDIRECT,Y
%11010010 @322 210 $D2 UNUSED
%11010011 @323 211 $D3 UNUSED
%11010100 @324 212 $D4 UNUSED
%11010101 @325 213 $D5 CMP ZERO PAGE,X
%11010110 @326 214 $D6 DEC ZERO PAGE,X
%11010111 @327 215 $D7 UNUSED
%11011000 @330 216 $D8 CLD IMPLIED
%11011001 @331 217 $D9 CMP ABSOLUTE,Y
%11011010 @332 218 $DA UNUSED
%11011011 @333 219 $DB UNUSED
%11011100 @334 220 $DC UNUSED
%11011101 @335 221 $DD CMP ABSOLUTE,X
%11011110 @336 222 $DE DEC ABSOLUTE,X
%11011111 @337 223 $DF UNUSED
%11100000 @340 224 $E0 CPX IMMEDIATE
%11100001 @341 225 $E1 SBC INDIRECT,X
%11100010 @342 226 $E2 UNUSED
%11100011 @343 227 $E3 UNUSED
%11100100 @344 228 $E4 CPX ZERO PAGE
%11100101 @345 229 $E5 SBC ZERO PAGE
%11100110 @346 230 $E6 INC ZERO PAGE
%11100111 @347 231 $E7 UNUSED
%11101000 @340 232 $E8 INX IMPLIED
%11101001 @351 233 $E9 SBC IMMEDIATE
%11101010 @352 234 $EA NOP IMPLIED
%11101011 @353 235 $EB UNUSED
%11101100 @354 236 $EC CPX ABSOLUTE
%11101101 @355 237 $ED SBC ABSOLUTE
%11101110 @356 238 $EE INC ABSOLUTE
%11101111 @357 239 $EF UNUSED
%11110000 @360 240 $F0 BEQ RELATIVE
```



```
%11110001 @361 241 $F1 SBC INDIRECT,Y
%11110010 @362 242 $F2 UNUSED
%11110011 @363 243 $F3 UNUSED
%11110100 @364 244 $F4 UNUSED
%11110101 @365 245 $F5 SBC ZERO PAGE,X
%11110110 @366 246 $F6 INC ZERO PAGE,X
%11110111 @367 247 $F7 UNUSED
%11111000 @370 248 $F8 SED IMPLIED
%11111001 @371 249 $F9 SBC ABSOLUTE,Y
%11111010 @372 250 $FA UNUSED
%11111011 @373 251 $FB UNUSED
%11111100 @374 252 $FC UNUSED
%11111101 @375 253 $FD SBC ABSOLUTE,X
%11111110 @376 254 $FE INC ABSOLUTE,X
%11111111 @377 255 $FF UNUSED
```

6502 MONITOR AND ANALYSER

By Martin Lewis and Andrew Foord

INTRODUCTION

The Monitor is an essential tool in the debugging of machine code programs. It has all the facilities of a normal monitor plus many new features. The standard commands allow memory to be examined, searched, edited, moved or filled; programs can be run at normal speed; breakpoints set and machine code disassembled to the screen or printer. Additional features include slow running of programs, single stepping and the Analyser. The Analyser allows conditions to be trapped when running a program; you may, for example, want to know when in your program a certain area of memory is written to, or when a register takes on a particular value. From the analyser you can define conditions and then when running machine code the analyser will inform you when any of these have been fulfilled.

TAPE MAP

Side A : The standard Monitor/Analyser program which resides at 3000 decimal.

Side B : A relocatable copy of the Monitor/Analyser program.

1. OPERATING INSTRUCTIONS

1.1 Using the Standard Version

The standard version is located at 3000 (decimal) and is approximately 12500 bytes long.

TAPE: The monitor can be found on tape 1, side A. Press **SHIFT** and **RUNSTOP** to load.

DISC: To load the standard version of the monitor from disc, use the following command:

```
LOAD "GENMON",8,1
```

and it will run automatically.

1.2 Using the Relocatable Version

The relocatable version of the monitor allows the monitor to be loaded into any part of the memory from 3000 (decimal) to 40704. Once loaded, the program will ask you where you would like it to be located. Enter a decimal address in the range 3000 to 40704.

TAPE: The relocatable version can be found on tape 1, side B. Use **SHIFT** and **RUN/STOP** to load.

DISC: The disc version of the relocatable monitor can be loaded using:

```
LOAD "RELMON",8,1
```

and it will run automatically.

NOTE: When the monitor is entered the BASIC ROM is automatically paged out. When the EXIT command is executed, the ROM is paged in again before returning to BASIC. If your relocated copy of the monitor (which has length 12300 bytes) lies partially under the ROM, you should not execute ROM ON.

The 256 bytes immediately after the monitor are utilised by the monitor for temporary stack storage.

2. SCREEN LAYOUT

The screen is split into four different windows:

1. The top right hand corner window displays the current state of the 6502 registers. The top line gives the value of the program counter and a disassembly of the instruction at that address. The next line of information shows the status flags (N:sign, Z:zero, C:carry, I:interrupt, D:decimal, V:overflow). Next the contents of the S, A, X and Y registers are displayed and the value of ADDR is given (this value will be discussed later). The top seven values on the machine stack are shown down the right hand side. Finally the state of the ROM (BASIC ROM $A000-$BFFF) is shown.
2. The top left window is used to show disassembly, the contents of the analyser stack and the printout of some other monitor commands.
3. Below the top two windows there is a Hex and ASCII dump of a section of memory, the area dumped is indicated by the value of the memory pointer, whose exact location is shown by a '> <' on the middle line of the dump.
4. Below this there is space for printing error messages and for inputting of commands.

Several commands clear the whole screen to print their results. At the end of displaying the information the user is asked to press a key and the screen will revert back to normal format.

3. THE EDITOR

The monitor's editor allows the input of up to 39 characters at a time on the bottom line of the display. The following keys can be used to edit the line:

cursor left/right	Move along line
DEL	Delete previous character.
INST (shift DEL)	Insert a space.
CTRL L	Clear the editing area.
RETURN	Execute the command in the editing area. If there is an error in the command, control is passed back to the editor with the cursor placed close to the error. Pressing RETURN with nothing in the editing area causes the display to be updated.

In the lower half of the screen there is a memory display. On the middle line of the dump there is a cursor (> <) which shows the current position of the memory pointer. The memory pointer can be moved around the memory using the following keys:

F5	Increment the memory pointer.
F6 (shift F5)	Decrement the memory pointer.
F7	Add 8 to the memory pointer.
F8 (shift F7)	Subtract 8 from the memory pointer.

The memory pointer can be set to a particular value using the MEM= command.

As stated before, the top left of the screen is usually used for displaying a disassembly and the following keys can be used to update this:

F1	Scroll the screen and show the next line (or if the top left has been cleared get the disassembly back again).
F3	Show the next page of the disassembly.

You can inspect the program at a particular address using the DISS command (see 4.1).

3.1 ENTERING COMMANDS

Numbers can be entered in one of four number bases as shown below:

Decimal	4785	
Hexadecimal	$12B1	(preceded by a '$')
Octal	@11261	(preceded by a '@')
Binary	%100101011001	(preceded by a '%')

A single ASCII character can be used to represent a number by enclosing it in quotes, e.g. "A" is equivalent to the number 65.

ON and OFF have the values 1 and 0 respectively and can be used instead of these numbers.

The monitor has a built-in calculator which means that all parameters can be entered as expressions which are evaluated with Reverse Polish arithmetic. Reverse Polish arithmetic and the calculator are discussed in detail in the analyser section of the manual. All that you need to remember is that the operator follows the operands:

e.g. 2 + 3
is replaced by 2 3 +

So to increment the memory pointer by 64 you would enter:

```
MEM= MEM 64 +
```

There are in fact numerous different operations available and you can include the values of registers and flags in your calculations.

4. MONITOR COMMANDS

The following notation is used for command syntax: parameters are enclosed in '<' and '>' type brackets and optional parameters are enclosed in '[' and ']'. For example, FRED <x> [,<y>] would mean that the command whose name is FRED can be followed by one or two parameters, one called <x> and the other an optional parameter called <y>. The meanings of the parameter names are given below.

<byte>	A number in the range 0 to 255.
<word>	A number in the range 0 to 65535.
<count>	As <word>.
<addr>	An address in the range 0 to 65535.
<start>	As <addr>.
<finish>	As <addr>.
<flag>	Either 0 or 1 (or ON or OFF).
<db number>	A DB number in the range 1 to 8.
<brk number>	A breakpoint number in the range 1 to 8.
<filename>	A string of up to 16 characters enclosed in quotes.
<byte list>	A series of numbers whose individual values do not exceed 255. A string of more than one character may be entered. Each character is treated as a separate number.

4.1 THE COMMANDS IN DETAIL

A=<byte>
X=<byte> e.g. X=6
Y=<byte>
S=<byte>
P=<byte>

Assigns a value to a register.

PC=<word> e.g. PC=$8000

Assigns a value to the program counter.

MEM=<addr> e.g.MEM=$C500

Sets the memory pointer to a particular value.

DATA <byte list> e.g. DATA "HELLO",13

Places a list of bytes at the memory location pointed to hy the memory pointer. '.' can be used as an abbreivation of the command DATA.

e.g. ."HELLO",13

EXIT

Returns to whatever called the monitor. As long as the monitor's workspace is not overwritten, all the options and breakpoints and other definitions set by the user will be retained. It should be re-entered with a SYS 3000 (or the new start address if you are using a relocated version).

NOTE: The BASIC ROM ($A000-$BFFF) is paged back on exit from the monitor.

FILL <start>,<finish>,<byte> e.g. FILL $4000,$5FFF,0

The block of memory from <start> to <finish> inclusive will be filled with the value of <byte>.

DUMP [<start> [,<finish>]] e.g. DUMP
or DUMP $C000
or DUMP $C000,$C100

Gives a HEX and ASCII dump of memory onto the screen. With no parameters the dump starts from the current value of the memory pointer and continues until the STOP key is pressed. With one parameter the dump starts from the specified address until the STOP key is pressed. With two parameters the dump continues until either the STOP key is pressed or the second address is reached.

LDUMP [<start> [,<finish>]]

This command is the same as DUMP except that the output is sent to the printer.

MOVE <start1>,<finish>,<start2> e.g. MOVE $1000,$1D00,$A000

The block of memory from <start1> to <finish> inclusive is copied into the area starting at <start2>. Note that the MOVE command is intelligent and so overlapping blocks will not cause corruption.

CHECK <start1>,<finish>,<start2> e.g. CHECK $1000,$1D00,$A000

This command compares two blocks of memory. The area from <start1> to <finish> is compared with the area starting from <start2>. If the two blocks are the same then the 'OK' message is printed, else 'Failed at <addr>' is printed, where <addr> is the address of the first mismatch.

SEARCH <start>,<finish>,<byte list> e.g. SEARCH $8000,$A7FF,$5A,$BB

The block of memory defined by <start> and <finish> inclusive is searched for the first occurrance of the specified string of bytes. If no match is found then 'string not found' is printed, otherwise 'Found at <addr>' is printed, where <addr> is the address of the first match. If a match is found then the memory pointer is set to the value of <addr>. To find the next occurrance use the NEXT command.

NEXT

The search specified by the most recent SEARCH command is continued so that the next occurrance of the string can be found. If no SEARCH command has been used before or the last search had failed to find any more matches, a 'No search string' error is given.

SEI

Sets the I flag (disabling interrupts).

CLI

Clears the I flag (enabling interrupts).

NOTE: When single stepping or slow running, interrupts are automatically enabled after each instruction, so do not try to single step code which performs the task of changing the interrupt vector. This portion of code must be executed at full speed.

LOAD <filename> [,<addr>]

The load command tries to load the specified file (if using TAPE the filename need not be given) and place it at the given address. If an address is not given the file will be loaded into the position it occupied before it had been saved.

SAVE <filename>,<start>,<finish>

The block of memory from <start> to <finish> is saved to the selected device.

JUMP [<addr>]

The machine code at the specified address is executed at normal 6502 speed. On entry to the code the actual 6502 registers are set to the values held by the monitor. Also the machine stack is copied down from workspace. If one or more breakpoints are defined then these are loaded into memory. The only way to return to the monitor is to use a breakpoint. (If no parameter is given then the value of PC is used).

CALL [<addr>]

This command is the same as the JUMP command except that a JSR (a call) is used to execute the machine code program so that an RTS can be used to return to the monitor. On returning to the monitor the registers are stored and the stack is copied back into workspace. Breakpoints are also recognised during a CALL command.

DISS <addr> e.g. DISS $1000

Disassembles to the window in the top left of the screen. (See the notes on the editor and F1 and F3).

LIST <addr> [,<finish>] e.g. LIST 3000,3500

This command disassembles to the full screen. If only one parameter is given then it will continue until the STOP key is pressed. If both are given then it will continue until either the STOP key is pressed or the <finish> is reached.

LLIST <start> [,<finish>]

Same as LIST except that output is also sent to the printer.

DB [<db number> [,<start>,<finish>]] e.g. DB
DB 3
DB 7,$8000,$800F

The DB command allows you to define sections of memory to be displayed as data instead of machine code. Up to 8 areas can he defined and are numbered from 1 to 8. With no parameters, the DB command displays the current selection of data areas. With one parameter, a number from 1 to 8, the specified data area is removed from the list. With three parameters, a data area can be defined.

5. DEBUG COMMANDS

5.1 Single Stepping

From the monitor's editor the following keys can be used to perform single stepping operations:

CTRL I	Increment the value of PC
CTRL D	Decrement the value of PC.
CTRL K	Skip the next instruction.
CTRL S	Single step the instruction at the current PC value and make PC point to the next instruction. If you have executed a SLOWCALL OFF then CALLs and JUMPs into the ROM will he executed at full speed.
CTRL E	As CTRL S except that JSRs and JMPs are executed at full 6502 speed.

SLOWCALL <flag>

This changes the operation of single stepping and slow running. If the flag is OFF then any calls or jumps to the operating system ROM are executed at full speed. If the flag is ON then the calls or jumps are single stepped as normal

NOTE: The SLOWCALL command does not affect CTRL E, or slow running in modes 4 to 7.

5.2 Slow Running

A slow run is an automatic single stepping. There are four different modes of screen display when slow running:

0. No screen update after each instruction is executed.
1. Only the memory dump is updated.
2. Only the register display is updated.
3. Both register and memory dumps are updated.

Slow running has several avantages:

1. The STOP key can be used to halt the program.
2. When debugging programs it is useful to see what is happening in slow motion.
3. Breakpoints can be set in ROM programs.
4. The analyser can be used to debug programs.
5. The TRACE command can be used to give a list of instructions executed.

There are eight modes of slow running, numbered 0 to 7:

Mode	Screen Update	Stepping Type
0	0	CTRL S
1	1	CTRL S
2	2	CTRL S
3	3	CTRL S

4	0	CTRL E
5	1	CTRL E
6	2	CTRL E
7	3	CTRL E

SLOW <byte>

Starts a program slow running from the address held in PC.

WORK=<start>,<finish>

The work command sets up a buffer where the addresses for the TRACE command are stored.

TRACE

During a slow run the value of PC for each instruction is stored in memory, the TRACE command prints out these instructions. The number of addresses remembered is the number that can be stored in the workspace defined by the user. If no workspace has been defined then no addresses will be saved. Each time a SLOW is executed the workspace is cleared.

The TRACE command starts from the oldest address that it can remember and works forward until it reaches the point at which the program stopped. The listing can he paused by pressing a key and restarted by pressing any other. Pressing the RUN/STOP key returns control back to the editor.

TRACE n

As above but only the last n instructions are printed.

LTRACE

Same as TRACE but output goes to the printer.

LTRACE n

As LTRACE but only the last n instructions are printed.

5.3 BREAKPOINTS

Breakpoints are special controls inserted into a program to make it change its mode of execution, it could cause the program to stop and return to the monitor (a normal breakpoint) or it could change the mode of execution to fast mode or one of the slow modes (a special breakpoint). Up to eight breakpoints can be defined at any one time.

Breakpoints are implemented differently depending on what mode of execution you are using at the time; there are two modes: slow running (automatic single stepping) and fast execution (JUMP or CALL commands).

5.3.1 Fast Execution

When running at full 6502 speed, breakpoints will only be recogised in RAM (since a machine code BRK instruction has to be put into the program). The BRK instructions are loaded into RAM just before execution begins and the original contents are put back when control returns to the monitor.

5.3.2 Slow Running

Breakpoints can be in ROM or RAM when slow running since nothing is actually placed into memory. Each time an instruction is run the value of PC is compared to the breakpoint addresses.

5.4 BREAKPOINT TYPES

There are 18 different types of breakpoints, all of them recognised in either slow or fast modes. The first type is the normal breakpoint which stops execution when it is encountered. The other 17 have special functions and are numbered 0 to 16. Types 0 to

7, when encountered, change the mode of execution to the slow mode corresponding to the breakpoint type. Types 8 to 15 are breakpoints with counters; each time a breakpoint is encountered the counter is decremented, and when the count reaches zero execution is halted. The table below shows the differences between them.

TYPE	EFFECT
0	Continue in slow mode 0
1	Continue in slow mode 1
2	Continue in slow mode 2
3	Continue in slow mode 3
4	Continue in slow mode 4
5	Continue in slow mode 5
6	Continue in slow mode 6
7	Continue in slow mode 7
8	Decrement count, halt if zero else slow mode 0
9	Decrement count, halt if zero else slow mode 1
10	Decrement count, halt if zero else slow mode 2
11	Decrement count, halt if zero else slow mode 3
12	Decrement count, halt if zero else slow mode 4
13	Decrement count, halt if zero else slow mode 5
14	Decrement count, halt if zero else slow mode 6
15	Decrement count, halt if zero else slow mode 7
16	Continue at full 6502 speed.

Breakpoints, once defined, can be turned on and off; a breakpoint in the off state is ignored if encountered.

Breakpoint type 16, when encountered, changes the execution mode from slow to fast. The instruction at the breakpoint is single stepped before the mode change is made, even when encountered in fast mode. This means that execution is slowed temporarily.

5.4.1 Encountering Breakpoints

If a breakpoint occurs and the program execution is stopped, the message "Breakpoint <number>" is printed (where <number> is the number of the breakpoint that caused the halt). PC is equal to the address at which the breakpoint occurred. To continue from a breakpoint either turn off the breakpoint and continue with a JUMP, CALL or SLOW, or if you require the breakpoint to stay active then single step past the breakpoint and then continue execution.

If a breakpoint is encountered and it cannot be identified (a BRK instruction in a program) then the message "BREAKPOINT <addr>" is given, where <addr> is the address at which the BRK instruction was encountered.

5.4.2 Breakpoint Commands

BREAK <brk number>,<flag>,<addr> e.g. BREAK 2,ON,$C000

Defines a normal breakpoint. Any previous definition will be lost. The flag indicates the state that the breakpoint will be left in after it has been encountered. Initially the breakpoint will be ON.

DEFBRK <brk number>,<type>,<addr> [,<count>]

Defines a special breakpoint. A count parameter is required only if the type is between 8 and 15, and this is the initial value of the counter.

Breakpoint types 0 to 7 and 16 always stay on after they have been executed. Types 8 to 15 also stay on as long as the count has not reached zero yet, when it does the breakpoint will be turned off.

BRK <brk number>,<flag>

Sets the state of the specified breakpoint to that of the flag. Turning one off will still keep the definition. Turning on a count type breakpoint will reset its counter to its initial value.

DELETE <brk number>

Deletes a breakpoint definition.

LBRK

Lists the breakpoints giving their type (either BRK for normal, SF for type 16, or number), their addresses and their current states (ON or OFF). Breakpoint types 8 to 15 also display initial and current count values.

6. INPUT/OUTPUT COMMANDS

DEC	Makes the monitor print addresses and numbers in decimal. HEX Makes the monitor print addresses and numbers in hexadecimal.
DISK	Make disk the selected storage device, and also read the status of the disk drive. The status is displayed on the error message line.
DISK "<command>"	Send a disk command to the disk drive, e.g: DISK "S0:name" to erase a file, or DISK "N0:name,ID" to format a disk, or DISK "V" to validate a disk.
TAPE	Make tape the selected storage device.
CBM	Tells the monitor to use a Commodore printer (default). CENTRONICS Tells the monitor to use a Centronics interface (if fitted to the user port).
CLR <flag>	If ON is given as the flag, then the screen will be cleared whenever a jump to a program is made, using JUMP, CALL or SLOW. If OFF is the flag then the screen will not be cleared. The default state (its state when the monitor is first loaded) is OFF.
LENGTH=	This sets the page length when using a printer with the Centronics port. If a value 0 is given (default) then no action is taken.
DIR	Read the directory of the disk drive (the DISK command must be used first to select the disk).

7. ERROR MESSAGES

OK	The last command was executed without errors.
COMMAND NOT KNOWN	The monitor can't understand this command.
NUMBER TOO BIG	A number is out of range.
BAD NUMBER	The monitor can't understand a parameter.
START>FINISH	<start> and <finish> parameters were entered with the <start> number greater than the <finish>.
BAD OPCODE	When single stepping or slow running the monitor has found an opcode which it cannot execute.
COMMAND ABANDONNED	This may occur if the stop key was pressed or an error occurred during SAVE/LOAD type commands.

STOP PRESSED	The STOP key has been pressed.
FOUND AT	See the SEARCH command.
FAILED AT	See the CHECK command.
NO SEARCH STRING	See the SEARCH command.
STRING NOT FOUND	See the SEARCH command.
BAD FILENAME	An illegal filename was given for a LOAD/SAVE/ DEFLOAD/ DEFSAVE command.
TOO MANY OPERANDS	Too many parameters are given after a command.
TOO FEW OPERANDS	Not enough parameters are given after a command.
DEVICE NOT PRESENT	This is given during a LOAD or SAVE command if the disk drive is not present (and has been selected) or if a disk error is encountered.
FILE NOT FOUND	Given during a LOAD/SAVE operation if the file can't be found.
BAD FILE	Given during DEFLOAD command if file is not found or it is of the wrong format.
READ ERROR	Given while reading from tape or disk if there is a loading error.
BREAKPOINT NOT DEFINED	
BREAKPOINT	Given when a breakpoint halts execution.
PROGRAM SPACE UNDEFINED	The user has defined no space for the Forth dictionary. (See PROG=).
OUT OF PROGRAM SPACE	There is no more room left for Forth definitions.
WORD ALREADY KNOWN	This Forth word already exists in the dictionary. (You will have to give your new word a different name).
NO DEFINITION	There is no definition after the ! or WORD command.
WORD NOT KNOWN	The interpreter can't find this Forth word in the dictionary.
STACK EMPTY	The Forth stack is empty and an operation needs a number.
STACK NOT EMPTY	The Forth stack contains more than one number after a STOP word has been interpreted.
DIVISION BY ZERO	
OUT OF STACK SPACE	The Forth stack is full.

8. THE ANALYSER

The analyser allows the definition of intelligent breakpoints which cause a program that is slow running to stop when a particular condition occurs. The conditions that can be monitored are: the state of the flags, values of registers, the contents of memory and if memory has been written to or read from.

Conditions are set up using a subset of the language called Forth (for those who already know Forth, the words already defined for the analyser Forth are listed at the end of this section).

Before anything can be done with the analyser, some space for user definitions must be reserved, this is done using the PROG=<start>,<finish> command (where <start> and <finish> define a block of memory to be used as program space). To start with about 250 bytes of program space will be sufficient, if the monitor has been loaded in at 3000 and is the only program in memory, the area from $4000 should be free, so use PROG=$4000,$4100.

8.1 Introducing Analyser Forth

The analyser uses a dialect of Forth as the breakpoint controlling language for three reasons. Firstly, a Forth compiler is easy to implement and does not require a lot of memory to run in. Secondly, the code generated is very compact. Thirdly, and most importantly, Forth programs execute very rapidly. This latter quality is essential if the analyser is to be of any practical use.

Those users who are familiar with Forth should have no problems with this dialect but should still read this section carefully. We do not recommend newcomers to the language purchasing a text on Forth because the analyser uses only a very small subset of the words and the examples in this section are aimed to provide sufficient tutorial.

8.2 Analyser Commands

EVAL

This will evaluate any Forth expression and then print out the contents of the stack in the top left hand corner of the screen.

Forth uses Reverse Polish arithmetic to carry out expression evaluation. What this means is that the operands are placed on the stack BEFORE the operator. This means that 3 + 4 would be replaced by 3 4 + and so on. Below are some examples.

Expression	Reverse Polish
2 + 5 * 4	2 5 4 * +
(2 + 5)	* 4 2 5 + 4 *
2 - 30 / 5	2 30 5 / -
(2 - 30) / 5	2 30 - 5 /

e.g. EVAL 3 4 + 2 6 * would print 7 12 (see section 8.3).

('?' can be used as an abbreviation for the EVAL command, e.g. ? 3 4 +).

PROG<start>,<finish> e.g. PROG= $8000,$80FF

Sets the area which is to be used for the storage of Forth words.

CLEAR

Clears the user dictionary, removing all the words.

ANALYSER <flag> e.g. ANALYSER OFF

Enables or disables the analyser from working during a slow run.

WORD <word name> <definition>
or ! <word name> <definition>

Defines an analyser word, <word name> must be made of alpha-numeric characters only and should not have been defined before. <definition> is at least one number or analyser Forth word.

LDEF

Print out all the user Forth word definitions.

PDEF

As above but output is also sent to the printer.

DEFSAVE <filename>

Store the user defined words onto the selected storage device.

DEFLOAD <filename>

Add the user words stored in the file to the words already held. (filename only optional when using TAPE).

EDIT <word name>

This command prints out the definition of the specified word, if known, and then enters the editor to allow you to change the definition. Entering the line by pressing RETURN causes the word to be changed to the new definition. If there is an error of some sort in the line then the old word is untouched. New storage space is used up for the new definition and not the space used for the old one, that space is ignored and wasted so eventually if you do a lot of editing you will run out of space. If this happens simply DEFSAVE all the words to tape or disk and use the CLEAR command followed by the DEFLOAD command to get the words back again. This clears out all the old definitions and rebuilds its list of correct definitions.

NOTE: If you try to EDIT a definition which expands to more than 40 characters the excess will spill onto the next line. This will cause problems for the editor since initially the cursor is left at the end of the second line but once it is moved on to the first line it cannot be moved back onto the second line again. It is rare for definitions to take up more than 40 characters, but it may help to perform editing of such lines in DEC mode instead of HEX since numbers are printed in less characters in DEC mode.

8.3 Using Analyser Forth

When processing a Forth definition the analyser works along the definition from left to right dealing with each number or operator in turn. Numbers are simply placed on the "stack". Operators are interpreted and the required function performed. "Stack" in the context of this discussion is the analyser stack, not the 6502 stack. How would the analyser evaluate the following string?

```
6 2 3 4 + * +
```

As each term is encountered, working from left to right, it is processed.

	Stack
6 is a number so it is placed on the stack	6
2 is a number so it is placed on the stack	6 2
3 is a number so it is placed on the stack	6 2 3
4 is a number so it is placed on the stack	6 2 3 4

+ is an operator so it is interpreted. The effect of the + operator is to take the last two stack entries, add them and put the result on the stack:

4 is removed from the stack	6 2 3
3 is removed from the stack	6 2
the sum 7 is placed on the stack	6 2 7

* is an operator so it is interpreted. The effect of the * operator is to take the last two stack entries, multiply them and put the product on the stack:

7 is removed from the stack	6 2
2 is removed from the stack	6
the product 14 is placed on the stack	6 14

+ is an operator so it is interpreted. The last two stack entries are removed, added and the result is placed on the stack:

14 is removed from the stack	6
6 is removed from the stack	
the sum 20 is placed on the stack	20

The EVAL command allows a line of analyser Forth to be executed immediately and shall be used to introduce the language to you. Forth uses a stack to store all its data, to get Forth to do anything for you, you must put data on the stack first, for example you might want to add the numbers 2 and 3 together using the EVAL command. First you must place the two numbers on the stack, to do this type:


```
EVAL 2 3 (and press ENTER)
```

The following will appear (as long as program space has been defined):

```
STATE OF STACK:

$3
$2

ENTER COMMAND
```

This shows you that the analyser has placed the two numbers on the stack. The stack is known as a first-in-last-out structure because you can keep adding items but if you want to get at something that was placed on the stack earlier on you must remove everything above it first. For example, say you place the numbers 1, 2 and 3 on the stack in that order. 1 is on the bottom of the stack and 3 is on the top; to get at 1 again you must first remove the 3 and then the 2. So numbers placed on the stack are read off in the opposite order to which they went on.

To add the two numbers together, Forth has a word called '+' which it understands to mean addition, it takes the top two stack entries and adds them together, placing the result on the top of the stack, so now type:

```
EVAL 2 3 +
```

and, hey presto, you get the answer $5 (note the analyser is printing the stack in hexadecimal - to change this to decimal printing enter DEC (& RETURN)).

To demonstrate how the stack works, try:

```
EVAL 1 2 3 +
```

This leaves 5 and 1 on the stack, showing that the two most recent entries are added together. Now try:

```
EVAL 3 2 1 - +
```

The answer should be 4; to see why, look at the diagram below (the 'state of stack' diagram shows the bottom of the stack to the left):

WORD	STATE OF STACK	EXPLANATION
EVAL	empty	clears the stack
3	3	puts 3 on the top of the stack
2	3 2	adds 2 to the top of the stack
1	3 2 1	adds 1 to the top of the stack
–	3 1	2 minus 1 equals 1
+	4	3 plus 1 equals 4

Note that '–' is another Forth word, this time meaning subtraction (you can find all the defined words listed at the end of this section).

Instead of putting numbers on the stack you could use the values of the 6502 registers, for example, EVAL X Y + will place the values of X and Y on the stack, add them together and put the answer back on the stack. Try the following:

```
X=100
Y=200
EVAL  X  Y +
```

The answer should be 300 (decimal). Executing a word that requires items on the stack when none are there will cause a "Stack Empty" error to be printed. For example, EVAL + will give such an error.

As mentioned above, +, –, X and Y are all defined words, you can define your own words by using the WORD command, for example:

```
WORD FRED DUP *
```

This defines a word called FRED which gives the square of the number held on top of the stack. DUP is a defined word which makes a copy of the top stack item onto the stack and '*' is 'multiply'. Note that when you enter this line nothing is executed, the

definition is just stored away in a program space. To list any definitions you have made and to see how much space you have left use the LDEF (list definitions) command. Now try:

```
EVAL 4 FRED 5 FRED
```

You should get the answers 16 and 25 (since 4 squared is 16 and 5 squared is 25); the diagram shows what happens:

WORD	STATE OF STACK	EXPLANATION
EVAL	empty	stack cleared
4	4	4 placed on stack
FRED→DUP	4 4	another copy placed on stack
→*	16	multiply the top two items
5	16 5	5 placed on stack
FRED→DUP	16 5 5	another copy placed on stack
→	16 25	multiply the top two items

The definition FRED can only be used after it has been defined, otherwise a 'word not known' error is given. Also FRED can't be redefined using another WORD command, it can only be altered using the EDIT command. Since word names could consist of numeric characters only it is possible (but not advisable) to redefine numbers; for example, try this:

```
WORD 10 11
EVAL 10 10 +
```

The answer printed will be 22 since you have redefined the number 10 to give the value 11.

To remove all definitions from the program space use the CLEAR command.

To change the definition of a word use the EDIT <word name> command.

STOP Definitions

To get the analyser to test for conditions while running a program, a stop must be defined. This is a word with the name STOP followed by a single digit from 0 to 9, e.g. STOP1 or STOP7. This STOP can be used like any other word definition but during SLOW running the stops are evaluated after every instruction has been executed. At the end of the STOP definition you should leave one number (usually a flag) on the stack, if this value is non-zero (true) the program will stop and the number of the STOP in which the halt was caused will be printed.

After a STOP definition the next instruction to be executed is displayed, not the one first executed, since PC always points to the next instruction to be executed.
To explain the use of STOPs we will need a demonstration program, type in the following list of commands:

```
MEM=$5000
DATA  $A0,$0,$A9,$1,$99,$0,$4,$18,$69
DATA  $1,$C8,$C0,$F,$D0,$F5,$60
```

This will load a short machine code program into the memory at $5000 (which should be free if the monitor/analyser is the only thing loaded) the disassembly of the code is shown below.

```
$5000: LDY #$0
$5002: LDA  #$1
$5004: STA  $400,Y
$5007: CLC
$5008: ADC  #$1
$500A: INY
$500B: CPY  #$F
$500D: BNE  $5004
$500F: RTS
```

Check that you have these instructions in memory by entering:

```
DISS  $5000
```


You should now see these lines on the screen, with some extra instructions at the end which we shall ignore.

The program puts characters onto the screen in the top left window.

To run this program type:

```
CALL  $5000
```

and you should get some characters displayed to the screen.

Suppose we want the routine to stop when the Y register contained a 3; to do this we need to define a STOP:

```
CLEAR              remove any old definitions
WORD STOP0  Y  3  =
```

STOP0 has now been set up to detect when Y = 3. The Forth word '=' compares the two numbers on the stack. If they are equal, 1 is put on the stack, otherwise 0 is put on the stack.

The analyser can only be used while SLOW running, not fast executing; to execute the program type:

```
PC=$5000
SLOW  3
```

Some characters will be displayed, then the monitor will print 'STOP NUMBER 0' and 'press any key'. When you press a key the monitor screen display will be updated.
In our second example we will introduce four important new words. ADDR, RD, WR and ACF are analyser words that allow the detection of reading or writing of memory (see the list of definitions). Using the same program as above type in the following definitions:

```
CLEAR
WORD SCREEN ADDR $0404 =
WORD STOP0 SCREEN WR &
```

The definition SCREEN, returns 1 (a true flag) if the last instruction accessed the memory location $0404 (a location in screen memory), otherwise it returns 0. The word & is used to ensure that the STOP only occurs if SCREEN is true AND (&) memory is being written to. Now run the program using:

```
PC=$5000
SLOW  3
```

The analyser will now stop at the CLC instruction (since the STA instruction previously had written to the location $404 and this is checked after this instruction had been executed). The analyser can also be used to check that a particular value is written to memory:

```
CLEAR
WORD VALUE ADDR C@ 6 =
WORD STOP0 VALUE WR &
```

The analyser will check if the value written to memory is 6 (screen code for 'F') and if so it will halt just after the 'F' had been put onto the screen.

There are several general points to note about the analyser:

1. The logical operator '&' and the bitwise operator 'OR' can be used to chain conditions together, for example "condition1 AND (condition2 OR condition3)" would be converted to Forth as:

   ```
   condition2 condition3 OR condition1 &
   ```

2. The WORD command can be abbreviated to !<word name> <definition>, for example:

   ```
   !FRED 1 1 +
   ```

3. Up to ten STOPs can be defined (STOP0 to STOP9) but it is worth remembering that the more definitions there are, the slower the program will run.

8.4 Analyser Forth Reserved Words

In the following list of definitions, a stack diagram is given for each word. This shows how the word affects the stack. In these diagrams the '>' sign splits the 'before' and 'after' parts. Numbers on the stack are represented by n1, n2 and n3; these are 16 bit integers and so are in the range 0 to 65535. Sometimes words use flags and f1, f2 and f3 represent these. A flag uses zero to represent false and a non-zero value (usually 1) to represent true.

Here is an example:

```
n1 n2 > f1
```

This would mean that the definition would expect at least two numbers on the stack, where n2 is the top stack item and n1 is the next stack item. After the word had been executed n1 and n2 would have been removed, leaving a flag (which has the value of 0 or 1) on the top of the stack. A 'STACK EMPTY' error is given if there are insufficient items on the stack when the command is executed.

8.5 DEFINED WORDS

8.5.1 Register Values:

PC	>	n1	This places the current value of PC onto the stack.
A,X,Y,S,P	>	n1	This places the value of the specified register onto the stack.
NF,ZF,CF,I,DF,VF	>	f1	This places the value of the specified flag onto the stack.

NF	Sign flag
ZF	Zero flag
CF	Carry flag
I	Interrupt flag
DF	Decimal flag
VF	Overflow flag

8.5.2 Using the 6502 Machine Stack from Forth

POP > n1

This word takes the last byte entered on the machine stack (converts it to a 16 bit form with msb=0) and puts it onto the Forth stack.

PUSH

This word takes the top number off the Forth stack and if it is less than 256 it puts this onto the machine stack as a single byte. If it is greater than 255 then a BAD NUMBER error is generated.

8.5.3 Memory Addressing

ADDR > n1

ADDR places the address of the last memory location accessed on the stack. If the most recent instruction did not write to, or read from, memory then ADDR will return the value 0. Not all memory accessed by an instruction is recorded; the following do not affect ADDR:

PHA, PHP, PLA, PLP, RTS, RTI, BRK

All of the addressing modes are recognised including those which use the registers.

ADDR can be used from the EVAL command and it will reflect the actions of the instruction PC is pointing to at that moment.

RD,WR,ACF > f1

All three instructions leave a flag on the stack. RD leaves a true flag (1) if the last instruction read from memory otherwise it leaves false (0). WR leaves a true flag if the last instruction wrote to memory. ACF leaves a true flag if the last instruction accessed memory (i.e. either read or write).

Some instructions read and write to the same location; they are:

Increments or decrements of a memory location
Rotations of memory locations

8.5.4 Arithmetic Operations

+	addition	n1	n2	> n3	(n3=n1+n2)
-	subtraction	n1	n2	> n3	(n3=n1-n2)
*	multiplication	n1	n2	> n3	(n3=n1*n2)
/	division	n1	n2	> n3	(n3=n1/n2)

These four mathematical operators use the top two stack items as their operands, if there are less than two numbers on the stack at the beginning of the word a 'Stack Empty' error is given. The operands are removed before the answer is placed on the stack.

Note that the analyser stores its numbers as 16 bit integers, i.e. a number from 0 to 65535 and so negative numbers and numbers greater than 65535 cannot be entered. The analyser has no overflow error and so only the least significant 16 bits of any answer is ever remembered, the following examples demonstrate this:

EVAL 65535 1 +	gives the answer 0
EVAL 0 1 -	gives the answer 65535
EVAL 500 500 *	gives the answer 53392

Since numbers are stored as integers, the result of division will be rounded down to a whole number, e.g. 10 4 / will give the answer 2 instead of 2.5.

8.5.5 Logic Operators

AND	bitwise AND	n1	n2	>	n3
OR	bitwise OR	n1	n2	>	n3
XOR	bitwise exclusive-OR	n1	n2	>	n3
NOT	bitwise complement	n1	>	n3	

These four operators perform the given bitwise logic operations on the top two stack items (only one for NOT) and after removing the operands the answer is placed on the stack. The logic operations are performed on the full 16 bits of a number and the tables below show the effect on the individual bits for the four operations:

0 AND 0 = 0	0 OR 0 = 0	0 XOR 0 = 0	NOT 0 = 1
0 AND 1 = 0	0 OR 1 = 1	0 XOR 1 = 1	NOT 1 = 0
1 AND 0 = 0	1 OR 0 = 1	1 XOR 0 = 1	
1 AND 1 = 1	1 OR 1 = 1	1 XOR 1 = 0	

8.5.6 Relational Operators

=	n1 equal to n2?	n1 n2 > f1
>	n1 greater than n2?	n1 n2 > f1
<	n1 less than n2?	n1 n2 > f1
>=	n1 greater than or equal to n2?	n1 n2 > f1
=<	n1 less than or equal to n2?	n1 n2 > f1
<>	n1 not equal to n2?	n1 n2 > f1
0=	n1 equal to 0?	n1 > f1
0>	n1 not equal to 0?	n1 > f1

These operators compare either the top two numbers on the stack, or the top stack item, with 0. f1 is either 1 or 0 to represent true or false. A 'Stack Empty' error is given if there are insufficient items on the stack. As with other operators the operands are removed from the stack before the answer is placed on the stack.

& Logical AND f1 f2 > f3

If the top two stack systems are both non-zero, a true flag (1) is put in their place, otherwise they are replaced by a false flag (0). The & operator can be used to chain conditions together, where any non-zero value on the stack represents a true condition and a zero value represents false. This word acts differently to the word AND since 1 2 AND would produce the answer 0, even though 1 and 2 both represent true conditions. The bitwise OR word can be used for chaining conditions together and 0= can be used to perform the logical NOT. For example you may want to convert the following to analyser Forth:

"Stop if condition1 is false and either condition2 or condition3 are true"

This is translated to:

(NOT condition1) AND (condition2 OR condition3)

and can be written in analyser Forth as:

STOP1: condition1 0= condition2 condition3 OR &

8.5.7 Other Operators

C@ byte fetch n1 > n2

The address on the top of the stack is replaced by the byte contained in that address.

@ word fetch n1 > n2

This word is like the C@ word except that two bytes are fetched from memory. E.g. to find the address held at 23100 (low byte) and 23101 (high byte) use: 23100 @ .

BIT bit test n1 n2 > f1

This word expects two numbers on the stack, n1 and n2. The result, f1, is a copy of the n2th bit of n1. n2 should be in the range 0 to 15 (since numbers are stored in 16 bits) and is taken in modulus 16 if greater than 15 (i.e. 31 represents bit 15 etc.).

8.5.8 Stack Operations

DUP n1 > n1 n1

Duplicates the top item on the stack.

SWAP n1 n2 > n2 n1

Swaps the top two stack items over.

OVER n1 n2 > n1 n2 n1

The second from top stack item is copied onto the top of the stack without removing it from its original position.

ROT n1 n2 n3 > n2 n3 n1

Rotates the top three stack items around as shown in the diagram.

8.5.9 Other Words

ON has the value 1 > 1

Places 1 on the stack.

OFF has the value 0 > 0

Places 0 on the stack.

MEM has the value of the memory pointer > n1

Places the value of the memory pointer on the stack.

@MEM > n1

Stacks the address at which the value of MEM is stored.

e.g. @MEM $61D3 !

would set MEM to $61D3.

UPDATE f1 n1 >

This tests f1 and if it is non-zero then the screen display (memory dump and register displays) is updated. n1 controls which windows are updated.

If n1 is 0 then nothing is displayed
if n1 is 1 then the memory dump is given
if n1 is 2 then the registers are displayed
if n1 is 3 then the memory dump is given and the registers are displayed

If n1 is greater than 3 then n1 MOD 3 is used.

CALL n1 >

This calls the machine code routine at address n1. This is provided so that you can add your own routines to the analyser easily. The X and Y registers are loaded with the address of a data area used by the analyser. (X contains the low byte and Y the high one).

The data areas contain the following:

0	Store for the A register
1	Store for the X register
2	Store for the Y register
3	Store for the P register
4	Store for the S register
5,6	Address of a routine to take a number off the Forth stack
7,8	Address of a routine to put a number onto the Forth stack

So these could, for instance, be accessed by:

```
STX  $19
STY  $1A
LDY  #location wanted
LDA  ($19),Y
```

You are given the addresses of routines which will pop and push numbers onto the Forth stack. Each number on the stack is stored as two bytes, so the high byte is stored in the X register and the low byte in the Y register.

e.g.
```
STX  $19
STY  $1A          ; store address of data area
LDY  #5
LDA  ($19),Y      ; get address of the 'getnum' routine
STA  getnum
INY
LDA  ($19),Y
STA  getnum+1

INY
LDA  ($19),Y      ; get address of the 'putnum' routine
STA  putnum
INY
LDA  ($19),Y
STA  putnum+1

; put number 426 onto the stack

LDX  #1
LDY  #170         ; 1 * 256 + 170 = 426
JSR  putnum

; get the number back

JSR  getnum

; the value 426 should be in the registers
```

```
        RTS
putnum  DEFB  $4C
        DEFW  0  ;          ; storage area for vectors to the 'getnum' and
                            ; 'putnum' routines
getnum  DEFB  $4C
        DEFW  0
```

IF f1 >

If the flag is true then the rest of the line is executed. If the flag is false then the rest of the definition is not executed.

! n1 n2 >

This stores the 16 bit number n1 at address n2.

C! n1 n2 >

This stores the 8 bit number n1 at the address n2.

?PAUSE f1 >

This waits for a key to be pressed if the flag is true, and continues once a key has been pressed. If the flag was false, no action takes place.

DROP n1 >

The top number on the stack is deleted.

NOP

This operation has no effect whatsoever. It is used for a blank definition, or if you want to slow a Forth word down slightly.

KEY > n1

Stack the ASCII value of the last key pressed. This is included to allow external control of your STOP definitions. If no key is pressed 0 is put onto the stack.

e.g: `KEY "A" = NOT IF TEST`

This would cause TEST to be executed unless "A" were pressed.

OPCODE > n1

n1 is the opcode value of the last instruction executed.

STACK > n1

n1 is the start address of the workspace used to store the machine stack. So STACK S + gives the address of the top of the machine stack.

8.6 EXAMPLE DEFINITIONS

Given below are a list of useful words that can be defined and then used to construct more powerful definitions (none of the definitions already exist in the analyser and so must be defined by you when you require them):

RANGE range checking n1 n2 n3 > f1

```
!RANGE ROT SWAP OVER >= ROT ROT <= &
```

Checks if the value n1 is in the range of n2 to n3 inclusive, for example `ADDR #8000 #80FF RANGE` checks if ADDR lies in the range #8000 to #80FF inclusive.

```
WORD   STATE OF STACK
RANGE  n1 n2 n3
ROT    n2 n3 n1
SWAP   n2 n1 n3
OVER   n2 n1 n3 n1
>=     n2 n1 f1
ROT    n1 f1 n2
ROT    f1 n2 n1
<=     f1 f2
&      f3
```


MEMWR Memory protection n1 n2 > f1

```
!MEMWR ADDR >= SWAP ADDR <= & WR &
```

This word can be used to protect areas of memory from being written to (or test if they are written to) by the machine code being slow run. n1 and n2 represent the start and finish addresses of an area of memory that is being protected. For example:

```
!STOP9 #8000 #8FFF MEMWR #0 #1FF MEMWR OR
```

This stop definition would cause the program to stop if it starts to write to the area from either #8000 to #8FFF inclusive or #0 to #1FF inclusive.

LXOR Logical exclusive-OR f1 f2 > f3

```
!LXOR 0> SWAP 0> XOR
```

The word LXOR is different to XOR since it treats the two numbers on top of the stack as flags (0false, 0<>TRUE). The logical XOR can be used to chain conditions, for example you might want a program to stop if either condition1 or condition2 were true but not if both are true, this would be written:

```
!STOP0 condition1 condition2 LXOR
```

8.7 Memory Details

There will be two versions of the Analyser/Monitor program supplied; one resides in a low area of memory and the other is a relocatable version.

The Analyser/Monitor uses the following zero-page locations:

```
$22,$23,$24,$25,$26,$27,$28,$29
```

They may be used by you in a machine code program when fast executing it, but do not use these locations when single stepping or when slow running.

8.8 The RESTORE Button

If you get into difficulties when using this program, even if one of your programs crashes when fast executing, the RESTORE button can be used as a 'panic' button. This button causes a non maskable interrupt (it means that the 6502 must stop what it is doing and go back to the start of the monitor).

WARNING: Sometimes some very strange things may happen if you use this facility. Only use this button if you have to! You can normally use the STOP key.

8.9 Analyser Examples

The following section shows how the Analyser can be used to track down bugs; it assumes that the area of memory around $8000 is free.

To demonstrate the analyser we need a program with a bug in it.

Example 1

Specification: The program given below (should) add up the ten numbers held in the bytes from $8000 to $8009 and place the total in the last element of the table.

```
$800A LDY #$0
$800C LDA #$0
$800E CLC
$800F ADC $8000,Y
$8012 INY
$8013 CPY #$A
$8015 BNE $800F
$8017 STA $8000,Y
$801A RTS
```

To enter this program into memory type the following:

```
MEM=$800A
DATA  $A0,0,$A9,0,$18,$79,0,$80,$C8
DATA' $C0,$A,$D0,$F7,$99,0,$80,$60
```

This program requires ten numbers in the locations $8000 to $8009; to place some data there, use:

```
MEM=$8000
DATA  1,2,3,4,5,6,7,8,9,0
```

To make sure that we have the program and data in the memory we can check it by listing it, but first define the memory from $8000 to $8009 as a data area using the command:

```
DB  1,$8000,$8009
```

and use this to list it:

```
LIST  $8000,$801B
```

The program can be run using:

```
CALL  $800A
```

The answer, which is 45 ($2D), should have been placed in the last location of the table, i.e. $8009 (the location originally containing 0). Examining $8009 shows that it still holds the value 0, a BUG! If we list the program using LIST we also notice that the program has become corrupted, the byte at $800A has changed from $A0 to $2D. Assuming we didn't know immediately why the program isn't working, we could use the Analyser to find out why the total is not being placed in $8009 and why the byte at $800A is being corrupted.

First define some program space.

```
PROG=$8100,$82FF
```

We could start out by trying to find out why the byte at $800A was corrupted.

NOTE:

In this case there is only one instruction which writes to memory, the instruction at $8017. Using the analyser illustrates how a similar bug in a much larger program could be found, where memory would be written to in many places and only one would be causing the bug.

The following definition will set up the analyser to detect when this byte is being written to:

```
!STOP1  ADDR  $800A  =  WR  &
```

Before we can run the program again the byte at $800A should be returned to its original value:

```
MEM=$800A
DATA  $A0
```

Now run the program again:

```
PC=$800A
SLOW  3 (the analyser only works in SLOW modes)
```

The program will stop with "press any key" as usual, but in addition gives the error message "Stop number $1" indicating that STOP1 has given a TRUE value. The instruction shown is RTS, so the instruction just executed was STA $8000,Y. This will tell us that the answer is being placed in the wrong place, $800A instead of $8009.

To see exactly what is happening we can single step the program but since the program loops around several times this might take quite a while, especially since we can assume that the addition part of the program is working. Suppose we want to single step from when the last byte of the table is read, up to there we can slow run the program.

The following definition sets up STOP1 to detect when the last byte in the table is read ($8009):

```
CLEAR
!STOP1  ADDR  $8009  =  RD  &
```

Now we run the program again (remember to correct the program first):

```
MEM=$800A
DATA  $A0
PC=$800A
SLOW  0
```

The program will now stop at INY since the last instruction read $8009. We can now use the single stepping to see what happens from now on. Using Control S single step past INY, CPY and BNE until you reach STA. You will note that Y=$A and so the address it will write to is (shown by the value of ADDR) $800A. This is because INY is executed at the end of the last run of the loop. We can't remove this so a DEY will have to be inserted just after the BNE. To do this enter:

```
MEM$8017
DATA  $88,$99,$0,$80,$60
```

Now run the program:

```
CALL  $800A
```

Afterwards examine the byte at $8009; it should be 45 ($2D), the sum of 1,2,3,4,5,6,7,8,9,0. Type:

```
LIST  $8000,$801C
```

and you will see that the program bas not been corrupted. Running the program again will give the result 90 ($5A) since the sum is now that of 1,2,3,4,5,6,7,8,9,45.

Example 2

The following example is included firstly because it is a fairly comprehensive example of the analyser's use, and secondly because it performs a test that some programmers may find useful at some stage.

Under normal circumstances a subroutine is exited with the stack in the same state as it was when the routine was entered. A common source of error (and one which is often difficult to trace) occurs when routines are called recursively. The definition in this example will cause a screen update and pause, if a return is encountered with the stack in a different state to that which was produced by the most recent call.

DEFINITION	DESCRIPTION
`!TABLE  $7003`	Sets the start of the table which holds the stack values.
`!PNT  $7000`	Sets the address for the pointer into the table.
`!INIT  PNT  TABLE  !  $7002  0  C!`	Initialises the table and table pointer.
`!INCPNT  PNT  DUP  @  1  +  !`	Increments the table pointer.
`!DECPNT  PNT  DUP  @  1  -  !`	Decrements the table pointer.
`!PRINT  1  3  UPDATE  1  ?PAUSE`	Updates the front panel display and waits for a key press.

```
!CHECK PNT @ C@ S <> IF PRINT
```

Fetches the last stack pointer value and compares it with the current stack pointer value. If they differ then the front panel display is updated.

```
!JSR OPCODE $20 = IF PNT @ S C!
INCPNT
```

Checks to see if the last instruction executed was a JSR. If so, then the current value of the stack pointer is entered into the table and the pointer is incremented.

```
!RTS PC C@ $60 = IF DECPNT
CHECK
```

If the current instruction is an RTS then the pointer is decremented and a CHECK performed to ensure that the stack is in the same state as it was when the last JSR was executed.

```
!STOP0 JSR RTS $0
```

Defines a stop condition that checks for matched calls and returns. Note the $0 which prevents termination. This means that the check will run indefinitely.

Using the Example Definition

The definitions given above can now be used to track down stack errors in your programs, but first note that before slow running any code with these definitions the Analyser program must first be initialised with:

```
EVAL INIT
```

Note also that the program will only work with the slowcall option turned on, so use:

```
SLOWCALL ON
```

before testing any programs.

To show the use of these definitions we have supplied an example program:

```
          .ORG  $9000

           LDA  #32          ; start with a space (' ')
LOOP       JSR  CHROUT       ; print a character
           JSR  PAUSE        ; wait a bit
           CLC               ; increment character counter
           ADC  #1
           CMP  #127         ; repeat until character 127 is reached
           BNE  LOOP
           RTS               ; te fin

PAUSE      PHA               ; use two nested loops to cause
           LDX  #64          ; a time delay
PAUSE1     LDY  #0
PAUSE2     DEY
           BNE  PAUSE2
           DEX
           BNE  PAUSE1
           JSR  GEYKEY       ; check if a key has been pressed
           CMP  #0
           BNE  PAWS         ; jump if a key has been pressed
           PLA
           RTS
```



```
PAWS       JSR  GETKEY       ; wait until another key has been
           CMP  #0           ; pressed and then return
           BEQ  PAWS
           PLA
           RTS

GETKEY     =$FFE4            ; Commodore ROM routines
CHROUT     =$FFD2
```

To enter this program from the Monitor type the following:

```
MEM=$9000
DATA  $A9,$20,$20,$D2,$FF,$20,$10,$90
DATA  $18,$69,1,$C9,$7F,$D0,$F3,$60
DATA  $48,$A2,$40,$A0,0,$88,$D0,$FD
DATA  $CA,$D0,$F8,$20,$E4,$FF,$C9,0
DATA  $D0,2,$68,$60,$20,$E4,$FF,$C9
DATA  0,$F0,$F9,$68,$60
```

As it stands the example program works, to see what it does type

```
CALL  $9000
```

The program prints a set of characters on the screen with a small time delay between each character. Pressing a key starts and stops the printing.

We will now introduce a bug. Type the following:

```
MEM=$902B
DATA  $EA
```

This modification to the program will change the PLA at the end of the 'PAWS' routine into a NOP. The program will now only work properly if the 'PAWS' routine is not used (i.e. no key is pressed). Pressing a key will cause a crash.

Say we wanted to use the Analyser to help track down the bug (assuming we didn't already know where it was). First reserve some analyser program space using:

```
PROG=$8000,$8400
```

and then type in the definitions given above. Use LDEF to check that you have typed them in correctly.

The example program uses two nested loops to cause a time delay, slow running this would take a long time. To speed this section of the program up while slow running we can use two breakpoints to make this piece of code (from $9011 to $9018) to be executed at normal 6502 speed. To do this enter the following:

```
DEFBRK  1,16,$9011
DEFBRK  2,0,$901B
```

We can now run the program under the Analyser:

```
PC=$9000
SLOWCALL  ON
EVAL  INIT
SLOW  0
```

Leave the program to run and you will see that characters are being printed (slowly). Now press a key, this will take us into the 'PAWS' routine where the computer will loop until another key is pressed. Pressing another key will cause the display to be updated. PC now points to the RTS at $902C, indicating that the stack is not set up properly for a return. This is where we would expect an error to occur since we now have no PLA instruction at the end of 'PAWS' to match the PHA at $9010. If we put the PLA back (first press RUN/STOP):

```
MEM=$902B
DATA  $60
```

and run the program again:

```
PC=$9000
EVAL  INIT
SLOW  0
```

The program will continue happily even if you press a key to stop and start the printing.

APPENDIX A - SUMMARY DF COMMANDS

COMMAND	PARAMETER	ACTION
A=	<byte>	Assigns value in <byte> to the A register.
ANALYSER	<flag>	Switch the analyser on or off according to the flag. (flag = ON or OFF).
BREAK	<brknumber>,<flag>,<addr>	Define breakpoint number <brk number>. If the address <addr> is met, the monitor will resume control. The breakpoint may be set to remain on or off.
BRK	<brknumber>,<flag>	The breakpoint <brknumber> is switched on or off according to the value of <flag>.
CALL		A call to the address in PC is made. Code executes at normal 6502 speed, but a return address is put onto the stack.
CALL	<addr>	As above but control is passed to the code at address <addr>.
CBM		Selects the CBM port as the printer port.
CENTRONICS		Selects the Centronics port as the printer port.
CHECK	<start1>,<finish1>,<start2>	Compares byte for byte, the contents of memory, lying between <start1> and <finish1> inclusive, with the area starting at <start2>.
CLEAR		Resets the program space and switches the analyser on.
CLR	<flag>	Defines screen state on running a program.
DATA	<byte list>	The byte list is placed into memory (abreviated to ".").
DB		Lists the current selection of data areas.
DB	<db number>	Deletes the specified data area.
DB	<db number>,<start>, <finish>	Defines a data area.
DEC		Sets number printing to decimal.
DEFBRK	<brk number>,<type>,<addr>	Define a SPECIAL breakpoint number <brk number>. If the address <addr> is met, the monitor will act according to the value of type.
DEFBRK	<brk number>,<type>, <address>,<count>	Define SPECIAL breakpoint number <brk number>. If the address <addr> is met then <count> is decremented. When <count> reaches zero the monitor will act according to the value of <type>.

DEFLOAD	"<string>"	Load the analyser definitions from the file called "<string>".
DEFSAVE	"<string>"	Save the current analyser definitions to a file named "<string>".
DELETE	<brk number>	Removes the breakpoint <brk number> definition.
DISK		Select the disk drive as current input/ output device, and read its status.
DISK	"<disk command>"	Send a command to the disk drive.
DISS	<addr>	Disassemble the contents of memory from address <addr> onwards.
DUMP		Output the contents of memory, starting at address MEM, to the screen until STOP is pressed.
DUMP	<start>	As above but start at the address specified by <addr>.
DUMP	<start>,<finish>	As above but finish if address <finish> is reached or if the STOP key is pressed.
EDIT	<word name>	Edit the specified analyser definition.
EVAL	<definition>	Evaluate the definition and print out the state of the analyser stack.
EXIT		Return control to the program that called the monitor.
FILL	<start>,<finish>,<byte>	The value <byte> is placed in the memory area ranging from <start> to <finish>.
HEX		Sets number printing to hexadecimal.
JUMP		Pass control to the machine code starting at the address in PC. Code will execute at normal 6502 speed. Unless breakpoints are encountered the monitor will not be re-entered.
JUMP	<addr>	As above but control is passed to the program at <addr>.
LBRK		List any defined breakpoints.
LDEF		List the user's analyser definitions.
LDUMP		As with the DUMP commands but output is sent to the printer.
LENGTH	<number>	Sets the printer page length.
LIST		Disassemble to the full screen starting at the address in PC.
LIST	<start>	As above but start at address <addr>.

LIST	<start>,<finish>	As above but stop when address <finish> is reached.
LLIST		As with the LIST commands but output is also sent to the printer.
LOAD	<file name>	Load the specified file to the address found in the header block.
LOAD	<file name>,<addr>	The file is loaded into memory to the address specified.
LTRACE		The contents of the TRACE memory will be listed to the printer.
LTRACE	<number>	As LTRACE but only the last n instructions are printed.
MEM=	<addr>	The memory pointer is set to the value of <addr>.
MOVE	<start1>,<finish1>,<start2>	Copy the contents of memory lying between <start1> and <finish1> inclusive, to the area starting at address <start2>.
NEXT		Finds the next occurrance of the search string in the memory range given in the last SEARCH command.
P=	<byte>	The Processor Status register is given the value of <byte>.
PDEF		As LDEF but output is also sent to the printer.
PROG=	<start>,<finish>	Define the memory between <start> and <finish> as the analyser program space.
ROM	<flag>	Page the ROM in or out.
SAVE	<filename>,<start>,<finish>	The contents of memory lying between <start> and <finish> is saved to a file.
SEARCH	<start>,<finish>, <byte list>	The block of memory specified is searched for the <byte list> given.
SLOW	<slow mode number>	Start a program slow running with update of the monitor display according to the value of the mode number.
SLOWCALL	<flag>	Determines the action of a ROM CALL when slow running.
S=	<byte>	The stack pointer is given the value of <byte>.
TAPE		Select the tape recorder as the current input/output device.
TRACE		The contents of the TRACE memory are displayed to the full screen.
TRACE	<number>	As TRACE but only the last n instructions are displayed.

WORD	<word name> <definition>	Define an analyser word.
WORK=	<start>,<finish>	Sets aside a workspace for TRACE.
X=	<byte>	The X register is given the value of <byte>.
Y=	<byte>	The Y register is given the value of <byte>.
.		As DATA.
?		As EVAL.
!		As WORD.

ASSEM8LER INDEX 64-MAC

MONITOR ANALYSER INDEX

UPDATE	52
WORD	44, 63
WORK=	40, 63
X=	37, 63
Y=	37, 63
@MEM	51
?PAUSE	53
&	50
@	51
!	53, 63
?	63
.	63

NOTES